LA-UR- 02-6573

Approved for public release; distribution is unlimited.

Title:

Design, Implementation, and Validation of Low- and Medium-Fidelity Network Simulations of a 30-TeraOPS System

Author(s):

Francis J. Alexander, Kathryn Berkbigler,
Graham Booker, Brian Bush,
Kei Davis, Adolfy Hoisie,
Nicholas Moss, Steve Smith,
Thomas P. Caudell, Donald P. Holten,
Kenneth L. Summers, Cheng Zhou

Submitted to:

world wide web

Los Alamos
NATIONAL LABORATORY

Design, Implementation, and Validation of Low- and Medium-Fidelity Network Simulations of a 30-TeraOPS System

Francis J. Alexander, Kathryn Berkbigler, Graham Booker, Brian Bush,
Kei Davis, Adolfy Hoisie, Nicholas Moss, and Steve Smith
Los Alamos National Laboratory
Los Alamos, NM 87545

Thomas P. Caudell, Donald P. Holten, Kenneth L. Summers, and Cheng Zhou
Albuquerque High Performance Computing Center
University of New Mexico
Albuquerque, NM 87131

30 September 2002

Abstract

The magnitude of the scientific computations targeted by the US DOE ASCI project requires as-yet unavailable computational power, and unprecedented bandwidth to enable remote, real-time interaction with the compute servers. To facilitate these computations ASCI plans to deploy massive computing platforms, possibly consisting of tens of thousands of processors, capable of achieving 10-100 TeraOPS, with WAN connectivity from these to distant sites. For various reasons the current approach to building a yet-larger supercomputer—connecting commercially available SMPs with a network—may be reaching practical limits. Better hardware design and lower development costs require performance evaluation, analysis, and modeling of parallel applications and architectures, and in particular *predictive* capability. We outline an approach for simulating computing architectures applicable to extreme-scale systems (thousands of processors) and to advanced, novel architectural configurations, and describe our progress in its realization. The simulation environment is intended to allow (i) exploration of hardware/architecture design space; (ii) exploration of algorithm/implementation space both at the application level (e.g. data distribution and communication) and the system level (e.g. scheduling, routing, and load balancing); (iii) determining how application performance will scale with the number of processors or other components; (iv) analysis of the tradeoffs between performance and cost; and, (v) testing and validating analytical models of computation and communication. Our component-based design allows for the seamless assembly of architectures from representations of workload, processor, network interface, switches, etc., with disparate resolutions, into an integrated simulation model. This accommodates different case studies that may require different levels of fidelity in various parts of a system. Our current implementation, includes low- and medium-fidelity models of the network and low-fidelity and direct execution models of the workload. It supports studies of both simulation performance and scaling, and the properties of the simulated system themselves. Ongoing work allows more realistic simulation and dynamic visualization of ASCI-like workloads on very large machines.

Contents

1 Introduction

This report outlines our goals and the progress we have made towards their realization as of the fourth quarter of FY'02. We have parallel efforts underway in the following areas:

- improving the portability, flexibility, efficiency, and robustness of the underlying simulation framework;
- developing relevant workload models, both direct and statistical;
- developing relevant network models, currently low and medium fidelity;
- validating the models against real hardware and applications; and,
- developing a visualization capability for use in both data comprehension and debugging.

This section outlines our goals and our approach to meeting them. Section 2 is an evaluation of the DaSSF discrete-event engine chosen for the simulations. Section 3 describes how workloads may be represented—exactly, or to varying degrees of approximation. Section 4 describes specific network topologies and a technique for their succinct description. Sections 5 and 6 describe the design and implementation of the low- and medium-fidelity network representation, respectively. Section 7 provides details on how large networks and simulations of them are visualized. Section 8 concludes.

1.1 Motivation

The magnitude of the scientific computations targeted by the US DOE ASCI project requires as-yet unavailable computational power, and unprecedented bandwidth to enable remote, real-time interaction with the compute servers. To facilitate these computations ASCI plans to deploy massive computing platforms, possibly consisting of tens of thousands of processors, capable of achieving 10-100 TeraOPS, with WAN connectivity from these to distant sites. For various reasons the current approach to building a yet-larger supercomputer—connecting commercially available SMPs with a network—may be reaching practical limits.

Better hardware design and lower development costs require performance evaluation, analysis, and modeling of parallel applications and architectures, and in particular *predictive* capability. Performance studies are routinely used to select the best architecture or platform for a given application, select the best algorithm for solving a particular problem, and to study scalability with respect to problem and platform size. Evaluating and analyzing the performance is challenging primarily because of the large number of components making up such systems and the complex dynamic interactions between them.

The tools of the trade in performance modeling and analysis are typically categorized as algorithmic/analytical analysis, statistical analysis, analysis with queuing theory, and simulation. Depending on the problem, one or more or these methods will be more appropriate than others. Although significant results have been obtained in recent work for an important class of applications of interest to ASCI [18, 17], analytical modeling of systems and applications of this scale is not always possible. Queuing models generally lead to very complex nonlinear equations whose solution is intractable. For systems of ASCI-proposed size and complexity *simulation* remains the predictive tool of choice, though simulation may be fruitfully augmented by analytical and statistical analysis.

Three related targets for our simulation effort have been identified: simulation of ASCI-scale parallel systems using a realistic ASCI workload, simulation of ASCI-scale storage systems and

I/O, and simulation of the high-performance ASCI wide-area network. All of these aspects of ASCI system design are equally important and tractable by the approach we propose. However, given the scale of the effort required, we are taking a staged approach to addressing these problems.

In conjunction with the other methodologies, the proposed simulation environment could be used for

- exploration of hardware/architecture design space;
- exploration of algorithm/implementation space both at the application level (e.g. data distribution and communication) and the system level (e.g. scheduling, routing, and load balancing);
- determining how application performance will scale with the number of processors or other components;
- analysis of the tradeoffs between performance and cost;
- testing and validating analytical models of computation and communication such as **LogGP** [10] and **BSP** [15].

A canon of the field of performance analysis is that hardware and software performance are inextricable—hardware performance is meaningful only in the context of applications—thus these capabilities are not entirely independent.

1.2 Goals

The *à la carte* project seeks to develop a simulation-based analysis tool for evaluating massively-parallel computing platforms including current and future ASCI-scale systems. Such a tool would provide a means to analyze and optimize the current systems and applications as well as influence the design and development of next-generation high-performance computers. Hence our general goal is to design and implement a *flexible* and *modular* simulation framework for design and analysis of extreme-scale parallel and distributed computing systems, and as an ongoing part of this process to validate the accuracy of results characterized by any particular model. An intermediate goal is to model, and validate the model of, the ASCI Q machine [19] with a realistic ASCI workload.

We take as given that it is not feasible to simulate an extreme scale machine and workload with perfect fidelity; on the other hand in certain circumstances it may be desirable to simulate some subset of such a machine with near-perfect fidelity. In any case the simulator itself should be able to exploit a machine of arbitrary size. Once defined, representations of logical *components* (e.g. processors or SMPs, network switches and interconnects, programs or workloads, etc.) should be easily assembled into differing configurations. Portability is essential. In more detail, the simulation system should

- be scalable to model systems comprising 10,000 processors or more;
- allow arbitrary sets of components to be represented with arbitrary degrees of fidelity, in terms of both structure (e.g. comprising distinct subcomponents) and timing;
- allow arbitrary (meaningful) configuration of components;
- allow description of the machine configuration to be as independent as possible of the descriptions of the components;

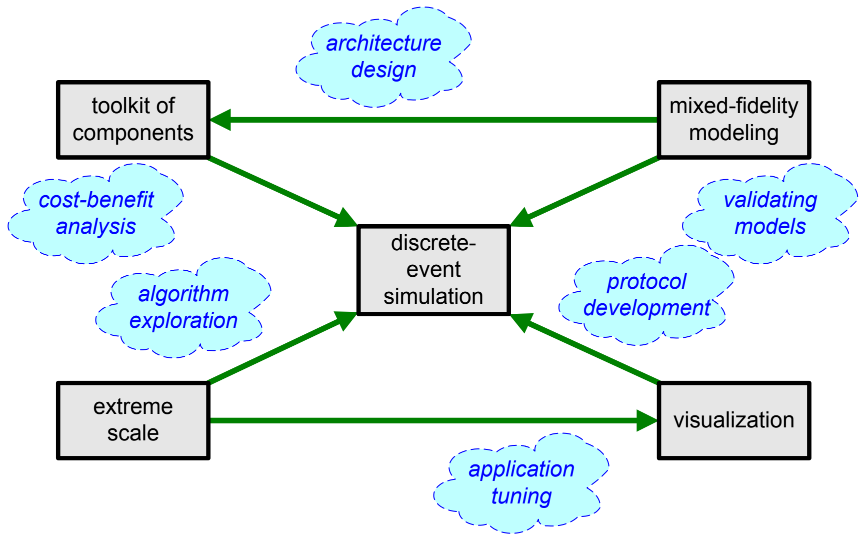

Figure 1: Goals and applications of the *à la carte* project: The boxes represent the project goals and the 'clouds' represent applications for the simulation and analysis tool.

- be genuinely portable across platforms ranging from single-processor workstations to clusters of SMPs;
- be able to interface to other distinct applications such as direct execution simulators and visualization systems.

Figure 1 graphically illustrates these goals and the applications they support. These requirements suggest factoring the simulator into three parts: component descriptions, configuration descriptions, and an underlying, reasonably generic, and reasonably light-weight simulation system with which all porting issues are associated. An object-oriented approach facilitates these goals.

It is clear that simulating systems of the size and complexity that we envision will require the use of parallel simulation [14]; ideally the simulation system would support distributed and shared memory simultaneously, i.e. fully exploit clusters of SMPs. Furthermore, the parallel simulation substrate must support composition of simulations and be very efficient in its implementation. We concluded that a conservative synchronization scheme would have the best chance of success for this application.

1.3 Approach

Our basic approach relies on an iterative development process for constructing components of appropriate fidelities and integrating them into a portable and efficient parallel discrete-event simulation that is scalable to thousands of (simulated) computational nodes. Components may be processors, switches, network interfaces, or application workloads, for example. Studies of hardware architectures are made by running our simulation for a particular aggregate system composed of these components. The output of the simulation captures the behavior and performance of the components, and may be visualized using techniques discussed in Section 7. Figure 2 illustrates the architecture of our simulator.

We chose a portable, conservative synchronization engine, the Dartmouth Scalable Simulation Framework (DaSSF) [9, 21], Dartmouth College's implementation of the Scalable Simulation Frame-

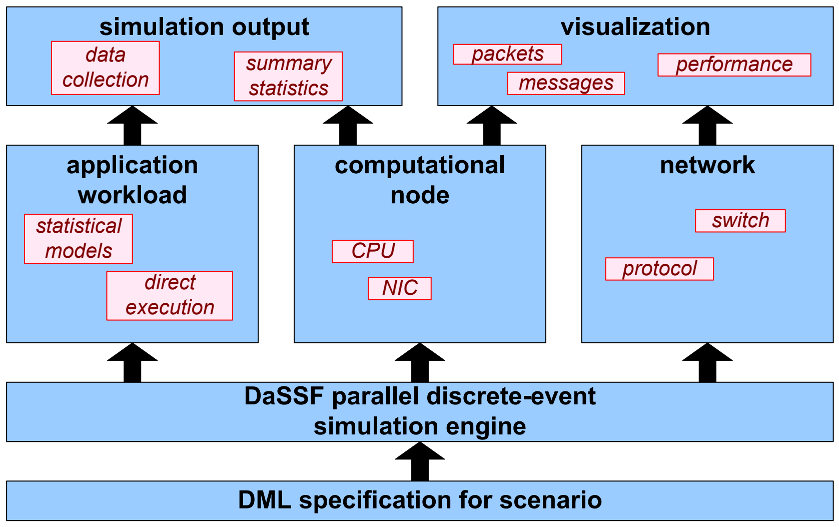

Figure 2: The architecture of the *à la carte* simulator: Simulation scenarios are represented using DML (Domain Modeling Language) and managed by the DaSSF simulation engine [21]. The application workloads, computational nodes, and networks are represented by software components that are assembled and connected according to the particular scenario being simulated. The results of the simulation may be studied visually, statistically, or in detail.

work API [37], for the handling of discrete events. DaSSF manages the synchronization, scheduling, and delivery of events in the simulation; it has a lean C++ API and supports both shared-memory and distributed-memory parallelism. We use the Domain Modeling Language (DML) to specify the architecture and workload to be simulated. DML allows one to easily construct libraries of reusable component specifications. The lower two levels of the architecture in Figure 2 comprise DML and DaSSF.

Our component-based design (represented by the middle layer in Figure 2) allows for the seamless assembly of architectures from representations of workloads, processors, network interfaces, switches, etc., with disparate resolutions, into an integrated simulation model. One can mix and match components of different fidelities to construct a model with the appropriate level of detail for a particular study. We are focusing on the development of a simulation capability that scales to tens of thousands of processors and that can execute on a wide variety of computing platforms.

The representation of application workload (shown on the left side of the middle layer in Figure 2) forms an especially important part of the simulation. Applications and computational workloads may be represented at a variety of fidelities. Each approach below addresses tradeoffs between the accuracy of the model and the computing resources required for the model: simple *random processes* can load the hardware with message traffic having specified statistical properties. These can be engineered to match the distribution of messages in a real application and can include temporal and spatial correlations between messages, but ignore some of the data dependencies. *Direct-execution* techniques allow one to run programs nearly exactly on real processors coupled to a simulated network. These are faithful to the actual timing of an application on a processor, but may be very computationally intensive and slow. From time series of fine-grained simulations we are using learning algorithms to construct *reduced models* of the full system dynamics. This involves regression techniques like neural networks or dimension reduction methods such as the Karhunen-Loeve expansion.

The collection of simulation output (at the upper-left in Figure 2) is vitally important for understanding the behavior and performance of the simulated system. Our approach is to permit the collection of information on all events (message sends, packetization, switching, etc.) present in the simulation at the finest level of detail. Because of the potentially voluminous nature of such data, we allow for filtering capabilities so that only data of interest will be collected in a given study. Statistical summaries also provide concise views of system performance and behavior. We are also pursuing visualization of these simulations (at the upper-right in Figure 2). We focus on both visualizing the execution of the simulation and on visualizing the performance of the simulated system.

The current *à la carte* implementation comprises low- and medium-fidelity models of a network and low-fidelity and direct execution models of workload. The low-fidelity models scale to at least 4096 computational nodes in a fat-tree network. This implementation supports studies of simulation performance and scaling, and also the properties of the simulated systems themselves. Ongoing work in our iterative development approach aims to improve the fidelity of the representations and protocols with validation at each stage. Future work will further emphasize validation, the representation of I/O and storage, and wide-area networking.

2 Evaluation of DaSSF for Use in Parallel Architecture Simulation

Dartmouth SSF (DaSSF) [20, 9, 26, 21, 12] is a C++ implementation of the Scalable Simulation Framework (SSF) [9, 37]. We have completed an evaluation of DaSSF for use as the underlying parallel discrete-event-handling substrate for the *à la carte* project. Identifying a high-performance and scalable discrete-event handling mechanism is critical for the project's success. While there may be other parallel simulation frameworks that could also meet our needs, we have focused on DaSSF and this section presents our evaluation of its suitability for our project.

2.1 Characterization of DaSSF

DaSSF is a parallel discrete event simulator that uses conservative simulation protocols to synchronize execution on multiple processors. It is the only SSF implementation that runs on both shared memory and distributed memory processors or a combination thereof. The distributed memory implementation uses MPI for communication between processors.

DaSSF is intended to be a high-performance and scalable simulator [21]. It achieves this in part by using a custom threading mechanism that handles memory very efficiently. This leads to the requirement to annotate the source code in particular ways to support the implementation of the thread behavior.

The SSF API provides five base classes that applications may subclass. The **SSF_Entity** class defines the entities in the simulation and maintains their state information. The **SSF_Process** class defines the behaviors that entities possess. Entities are connected to each other via channels, an **SSF_OutChannel** in the transmitting entity and an **SSF_InChannel** in the receiving entity. An **SSF_Event** represents the information that flows between entities across the channels. Additionally, DaSSF enhances SSF by providing classes for random number generation, data collection, and simple statistics, and for modeling semaphores, timers, and direct event scheduling.

DaSSF models may be constructed from scripts written in the Domain Modeling Language (DML). A DML script is a recursively defined list of attributes which are key-value pairs. Special keywords are included to simplify model construction by supporting the object-oriented concepts of composition and inheritance. Parsing methods are provided, but the semantic interpretation of the attributes is left to the application. The use of DML is not required; models may also be constructed programmatically from the main function.

DaSSF is available for a variety of platforms including the SGI Origin 2000 cluster, the Compaq AlphaServer cluster, SUN Enterprise systems, clusters of Linux workstations, and more.

2.2 Evaluation Strategy

To determine the suitability of DaSSF as a parallel simulation substrate, a small prototype model was built. The purpose was to allow us to become familiar with DaSSF and to gain experience in constructing models. The prototype was to be a learning experience and feasibility study rather than the basis for conducting a specific simulation study. The prototype that was implemented is described in more detail in Section 5.

Our requirements for the prototype were that it exercise all the essential components of DaSSF and several of the DaSSF extensions to SSF. A small model that used all the features was desirable for rapid development. At the same time we were also interested in the scaling properties of DaSSF since we want to develop very large models in the future. We were not concerned with modeling our components with high fidelity, but rather with determining whether DaSSF was an appropriate substrate for developing components with arbitrary levels of fidelity and whether these components could be easily configured into arbitrarily large models.

We were also interested in the capability of integrating DaSSF with our own C++ classes as well as with standard components such as the Standard Template Library (STL). The data collection capabilities provided by DaSSF were another topic to be investigated. The ease of debugging simulations that use DaSSF was also to be evaluated.

DaSSF is provided for several platforms and may be compiled with the native compilers or with GNU g++. We tested our model on a variety of platforms using native compilers and vendor-optimized versions of MPI.

2.3 Using DaSSF

2.3.1 Design Limitations

A concept that is important in conservative synchronization methods is that of *lookahead.* If the earliest time that a logical process at time T can schedule an event for any other logical process is at time $T + L$, then L is known as the lookahead for the first process.

A DaSSF *timeline* (another name for a logical process) is a submodel that may run concurrently with other submodels [26]. Entities are assigned to timelines, and entities on the same timeline are said to be coaligned. Each timeline maintains its own event list from which events are processed in non-decreasing time-stamp order. Entities on different timelines communicate exclusively through messages passed over channels. Entities on the same timeline also communicate via messages but may additionally use other mechanisms. In DaSSF having a substantial lookahead is more important across timelines than within a timeline. Assignment of entities to timelines is the user's responsibility, and may have a large effect on simulation performance especially when the latencies on the channels (lookaheads) differ greatly.

A simple example will illustrate how the assignment of entities to timelines affects performance. Consider four entities A, B, C and D. A is connected to B, B is connected to C, and C is connected to D. The (simulated) latencies of the connections between A and B and between C and D are much longer than the latency between B and C. In this example A and D send messages to each other with an average delay between messages of 10^6 nanoseconds and with no more messages sent after 10^7 ns. The simulation ended at 10^8 ns which was sufficient for all messages to reach their

timelines	wall secs	timeline switches
1	0.055	4
2	40.4	11661158
3	0.36	75030
4	42.3	12015316

Table 1: Example of effect of number of timelines on DaSSF performance.

timelines	end time	wall secs	timeline switches
1	10^8	0.055	4
1	10^9	0.055	4
1	10^{10}	0.056	4
3	10^8	0.36	75030
3	10^9	3.08	754455
3	10^{10}	29.72	7347995
3	10^{11}	390.88	75000030

Table 2: Example illustrating that timeline synchronization occurs even in the absence of events.

destination. Table 1 presents the results for one timeline (all entities on the same timeline), two timelines (A and B share a timeline and C and D share a timeline), three timelines (B and C share a timeline), and four timelines (each entity on its own timeline). The table includes the runtime in seconds and the number of DaSSF timeline context switches. The runs were made on a single processor Solaris workstation.

DaSSF timelines synchronize with each other using a global synchronous barrier mechanism. This synchronization occurs until the end of the simulation even when the event lists of all timelines are empty. This effect may be seen in Table 2 which reports results for the same example described above, but this time for one and three timelines and increasingly longer simulation times. Note again that all messages were completed prior to time 10^8 ns. For a single timeline the runtime remains the same regardless of when the simulation ends, while the runtime increases with increasing end time using multiple timelines.

In DaSSF the behavior of an entity is defined in one or more **SSF_Process** instances. A process that contains computations interspersed with DaSSF wait statements which are used to advance simulation time is called a procedure. DaSSF supports two types of procedures, simple procedures and procedures which are not restricted. The execution path of a simple procedure must end with a **wait** statement. Such procedures are desirable because they are implemented more efficiently in DaSSF.

The requirements for annotating the source code are not too cumbersome, but do have to be meticulously observed because DaSSF contains little error checking in this area. The DaSSF User's Manual [21] describes the annotations, lists some limitations that derive from how the source code is parsed by the translator, and also warns of the importance of appropriately declaring **SSF STATE** variables.

DaSSF provides and manages an output collection mechanism that records output from entities distributed among any number of processors. Each processor has a single output file to which output is dumped in binary format. The **dumpData** function is a part of every **SSF_Entity**, which implies that output may only be written by existing entities. The user is permitted to define a global wrapup function which is called just before the simulation terminates. In this function one may retrieve the data from the multiple output files in time stamp order and organize and print it as desired. When the simulation terminates prematurely, such as due to an error, the data collected thus far is not post-processed by the wrapup function. This is unfortunate as the partial data could be a valuable clue to the location and cause of the abort.

In DaSSF 3.2.3 shared memory mapping functions and other portions of DaSSF code reliant on 32-bit system calls such as **lseek()** and **map()** limit the heap size of a single DaSSF process to approximately 2 GB. Similarly, lack of 64-bit addressing support for large files limits the maximum output data to 2GB. These two constraints pose a problem for *à la carte* models with sizes on the order of thousands of nodes and executed for large simulation times. The capability to run DaSSF in distributed mode is essential to overcome the memory limitation, and the output size limitation is overcome by clearing and streaming output buffers in chunks, and using the 64-bit file system support provided via OS file redirection.

The DaSSF extensions to SSF make modeling easier and we are using them despite the fact that this makes our models non-portable to other SSF implementations. Semaphores are more convenient than internal channels for signaling between processes of the same entity or coaligned entities. Timers provide a way to schedule a future action that can be cancelled at a later time if it is no longer desired. This is helpful because an **SSF_Event** cannot be canceled once it has been placed on the event list. DaSSF normally provides a process-oriented simulation world view, but it has functions that provide a discrete-event world view for situations where the extra performance gains outweigh the convenience of process-oriented simulation. We have successfully used semaphores and timers in our models. We are also using the DaSSF random number generation capabilities.

Most of the DaSSF constructs are implemented for distributed memory as well as shared memory architectures. However, two recently added features, user barrier synchronization and appointment channels, are only available for shared memory. Processes that coordinate through a semaphore must belong to entities that are coaligned in the same timeline, so these entities cannot be on distributed processors.

2.3.2 Model Building

DaSSF models may be constructed from DML files or programmatically. When DML scripts are used, the model topology is defined in the model DML file. Properties of model components, the number to be instantiated, and their connectivity are specified in this file. The machine DML file describes the hardware platform that the simulation will run on. The runtime DML file contains runtime information such as simulation start and end times and the names of the other DML files. DaSSF provides a partitioner that constructs the simulation components from the topology in the model DML file and assigns these components to the parallel computing platform.

The use of DML is not required. Models may be constructed programmatically from the **main** function. Thus far we have not experimented with this means for constructing models.

In Section 4.2 we describe our efforts in the development of a simple hardware description language to formally specify architectural configurations. This will facilitate the quick assembly of architectures for case studies where an appropriate DML input file might be cumbersome to construct.

2.3.3 Portability

We regularly build DaSSF and run small simulations on a single processor Solaris workstation. We usually use the g++ compiler, but the SUNpro compiler also works fine. Any problems encountered on this most basic platform often also occur on the more sophisticated platforms too.

Our experience with DaSSF on Intel workstations and clusters running Linux is also good. We generally have no problems compiling, linking, or running on these Linux-based systems.

Building DaSSF and running simulations on our SGI Origin 2000/IRIX cluster is more complex than on other platforms. This occurs in part because, unlike at Dartmouth, we must interact with the cluster via the Load Sharing Facility (LSF). We prefer to use the native MIPSpro compiler and MPI implementation for better performance and local support, and after numerous attempts have a combination of options that is consistent with features required by DaSSF. We have successfully run our simple statistical models on multiple processors on multiple boxes, and we have also run a short direct execution model on this platform. A recurring problem on this platform is that when a model finishes it sometimes does not properly interact with LSF to terminate the job, but rather continues to run until the LSF time limit is exceeded.

We were unable to successfully compile and run our simulation on Alpha clusters running the Tru64 operating system. Numerous problems were found using the compilers (both the native compilers and various versions of g++) and libraries on these machines. Problems include segmentation faults in the compiler, name space clashes between DaSSF and the standard libraries, and failure of ISO standards-compliant C++ code to compile.

DaSSF is not available from Dartmouth for the Alpha/Linux platform. However with consultation from the DaSSF developer we have successfully ported DaSSF to this platform. This effort, as well as work in progress on porting to the IA-64 platform, are described in more detail in subsequent sections.

2.3.4 Robustness

Early on we encountered several problems in trying to use DaSSF for our prototype, but the DaSSF developer was quite responsive in addressing our questions and fixing bugs that we uncovered. We were initially unable to use DaSSF and the STL together; this has been corrected. Our early model exhibited a lot of memory leaks. This was in part due to lack of clarity in the DaSSF manual regarding which objects DaSSF automatically destroys and which are the responsibility of the user. After eliminating those leaks, we believe the DML parser still contains memory leaks. We found that DaSSF works handily with sub-directories and namespaces, except that file names and event class names must be globally unique.

2.3.5 Debugging

We are able to debug our models using gdb on Linux machines, but have not fully succeeded in using gdb with DaSSF on Solaris where the technique that one uses to display variables on Linux does not work. Debugging is generally more cumbersome because the source-to-source translation that is needed to support the threading mechanism modifies the code and variable names. Reference to the translated code rather than the original source code is often required in order to understand the debug information.

2.4 Porting DaSSF to Linux/Alpha

DaSSF includes built-in support for distributed MPI execution on a limited set of platforms. Our 64-node Compaq ES40 Alpha cluster (*wolverine*) running Linux is in principle a superb platform for our purposes but is not fully supported by the current DaSSF distribution. DaSSF 3.2.3 includes full support for Alpha architectures running OSF (True64) but not Linux. Unsupported platforms are flagged as "generic" and build DaSSF binaries whose usage is limited; specifically, distributed and shared memory functionality are disabled.

A series of modifications and additions were applied to DaSSF 3.2.3 to enable distributed and shared memory support for Alpha-Linux. The changes include basic modification of Makefiles and configure scripts, shared memory mapping functions, POSIX threading support, GNU linker scripts, and cover other miscellaneous areas implementing DaSSF kernel functionality. Some portions of the DaSSF kernel implementation were easily adapted from their Alpha-OSF and X86-Linux counterparts, however, completion of SMP support proved more challenging. On fully supported architecture/OS combinations DaSSF provides the choice of two methods of using shared memory: POSIX threads or UNIX processes. A port of the POSIX threads approach was successfully completed but support of the UNIX process type is still under investigation: it requires at least several low-level modifications and coding at the Alpha-processor machine level. For our purposes, however, the POSIX threads method is sufficient, enabling effective use of the *wolverine* cluster. The port has been tested and executed with a variety of models, including several large time-scale executions using a 512-processor Uniform Workload model which were used for statistical analysis and characterization (Section 3.1).

Support for shared memory using the UNIX process mechanism remains to be completed. Complete instructions for reproducing the port, and the additional source files required, are archived elsewhere.

2.5 IA-64 Simulation Experiments

The development of Intel's IA-64 (Itanium, Itanium 2) and the proliferation of systems based on this architecture represents what may become a large player in the 64-bit general computing industry. At the same time the future of the HP Alpha is limited. So while the Alpha system *wolverine* is the most capable system available to us for simulation, and provides calibration data for modeling the future ASCI Q, to maintain currency and avoid obsolescence porting DaSSF and *à la carte* to promising newer architectures is a small but important part of our efforts.

The ability to employ a variety of architectures also makes possible more rigorous validation, refinement, and calibration of our conceptual and implemented models (Section 6.6).

Porting DaSSF and *à la carte* to IA-64/Linux has just begun. The DaSSF distribution and included examples build and run correctly in "generic" mode just as for Alpha/Linux. Thus the outstanding porting efforts include supporting shared memory using both of the standard DaSSF mechanisms.

The current IA-64 cluster available to us (*balli*) is equipped with a Quadrics network of the same type (Elan 3) as *wolverine*; this will further facilitate validation of the models and their parameterization.

2.6 Competing Simulation Frameworks

Although we have focused on DaSSF as our simulation substrate, we try to remain aware of other parallel discrete event simulation (PDES) environments that may be used for computer architecture simulation. A good overview of the PDES field was recently published [14]: references [4] and [22] contain surveys of languages and libraries for PDES.

SSFNet is a collection of Java SSF-based components for modeling and simulation of Internet protocols at and above the IP packet level of detail [37]. SSFNet is available in source form under the GNU public license, but it requires a Java SSF simulation kernel in order to run. A commercial implementation of Java SSF is available from Renesys Corp., and a license for research purposes is also available [37]. While we could not use SSFNet directly because it is written in Java, the common API between SSFNet and DaSSF may allow us to leverage our work with it when we reach the project stage of simulating the ASCI wide-area network.

The Wisconsin Wind Tunnel II [25], is a conservative synchronization parallel architecture simulator that utilizes direct execution to simulate multiprocessors with in-order processors and a simulated memory system. It is limited to the Sparc V8 instruction set. It appears that this simulator is no longer undergoing active development.

Researchers at UCLA have developed the Parsec language [3], and the COMPASS simulator for performance prediction of MPI programs. Parsec is a C-based parallel simulation language that supports both conservative and optimistic synchronization approaches. MPI-SIM [31], a component of COMPASS, is a library for the direct-execution driven simulation of MPI programs. It is built on top of a MPI-LITE, a library that supports multithreaded execution of MPI programs using a subset of MPI. Other components of COMPASS include a parallel I/O simulator and a parallel file system simulator [2].

A well known general purpose parallel discrete event simulation framework that uses optimistic synchronization techniques is Georgia Tech Time Warp (GTW) [11]. GTW is implemented for shared-memory multiprocessors and for heterogeneous networks of workstations, but only the shared-memory version is freely distributed [16].

Also available from Georgia Tech is Parallel/Distributed NS (PDNS) [30], a parallel implementation of the publicly available sequential *ns* simulator [7, 27]. The *ns* simulator is targeted toward networking research and includes several protocol modules. PDNS uses a conservative approach to synchronization. The current version of PDNS appears to be at least one version behind the latest release of *ns*.

3 Representing Workloads for ASCI Applications

The representation of the application workload is an important component in the *à la carte* framework. The framework is designed to allow multiple representations with varying degrees of fidelity. In this section we describe several workload representations that have been implemented.

3.1 Simple Statistical Workload

Our simplest statistical model for a workload is characterized by three random variables: an exponentially distributed delay between messages, an exponentially distributed message size, and the message destination, where all possible destinations are equally likely. The average values for message delay and size are specified in the DML model input. The destinations that each source node can send to may be specified individually for each node.

3.2 Direct Execution Workload

In this section we discuss a variant of the workload component that generates interconnection network traffic according to the demands of an actual running application. This representation of the workload, known as *direct execution*, is more accurate than using simple statistical distributions to generate network traffic. (Our work on attempting to derive complex statistical characterizations of workloads, described in Section 3.3, falls between these two extremes.)

In direct execution simulation the application is executed on the same machine used to perform the simulation. The application is typically modified to call the simulator only for those operations that differ between the host machine and the simulated machine. Using the host machine to directly

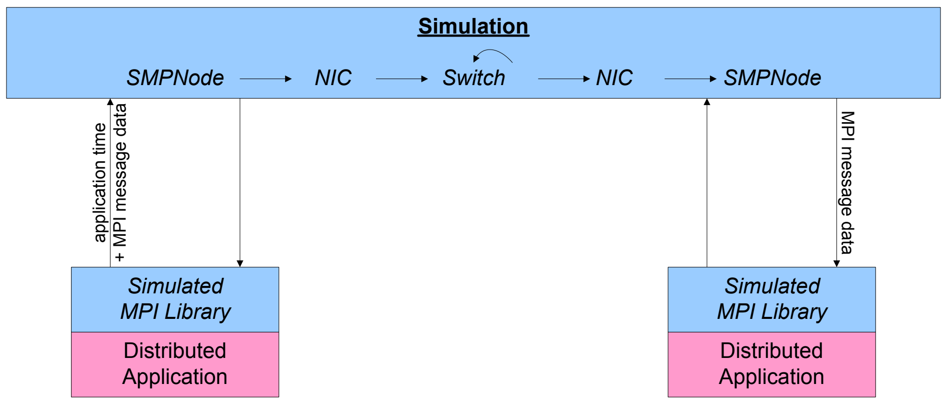

Figure 3: A distributed application directly coupled to a DaSSF-based simulation.

execute some instructions rather than simulating all instructions can result in considerably faster execution with minimal loss of accuracy when the host and target have similar architectures.

Our experiments with direct execution thus far have focused on simulation of communications on the interconnection network and direct execution of the computational aspects of an application. Most ASCI applications of interest use MPI (Message Passing Interface) for communication between parallel processes. We have developed a modified version of the MPI library that interfaces with a DaSSF simulation of an interconnection network. Figure 3 depicts the flow of information between the application and the simulation. Application calls to MPI are routed through the simulation rather than using the network available on the host machine. This is of course slower than using the native MPI implementation, but it allows us to study different types of interconnection networks via simulation. The final results from the application are the same as they would be without coupling to the simulation. The simulated runtime may be shorter or longer than the application runtime on the host machine depending on whether the simulated network is faster or slower than the host network.

The framework shown in figure 3 has been tested with SWEEP3D as the distributed application. A replacement MPI library has been developed for those MPI calls used by SWEEP3D. DaSSF processes have been developed to interface with the replacement MPI library. This involves launching the application, communicating with each distributed process through pipes, and handling each type of received message appropriately in the simulation. The modified MPI routines use timers to record the elapsed time since the last call to an MPI routine (i.e., the amount of time spent directly executing application computations). This timing information is sent along with the MPI message data to the object that represents the sending node in the simulator. The simulator adds the application time to the simulated time before forwarding the message through the simulation to the receiving node.

Obtaining accurate timing information for the directly executed code is critical to this approach. We desire a timing mechanism that has high resolution to be compatible with the simulation where time is reckoned in nanoseconds, and one that is available on a variety of platforms to allow maximum portability of the simulation. Table 3 summarizes the results of our investigation into the availability of timing software packages on platforms of interest to us. The entries in the table show the resolution of the timer. Where an operating system patch is required in order to use the software, the name of the patch is indicated.

All of these timing packages have drawbacks. The fact that most counters are not saved on context switches is serious for our purposes. PAPI appears to be the most promising and is still

Tool	Alpha TRU64	Alpha Linux	Pentium Linux	MIPS IRIX	UltraSparc SOLARIS
getrusage	μs	ms	10 ms	10 ms	10 ms
Atom [36]	cycle	X	X	X	X
PAPI [28] (patch)	cycle pfm[1]	cycle iprobe	cycle perfctr	cycle	cycle cpc
PCL [29] (patch)	cycle pfm	X	cycle perfctr[2]	cycle	cycle cpc/perfmon
utime [39] (patch)	X	?	μsec utime	X	X

pfm: *pseudo device driver, not configured into kernel by default*
iprobe: *kernel driver and library that must be loaded into kernel*
perfctr: *Linux kernel patch for 2.2.16 to 2.4.10*
cpc: *package for Solaris 8. PCL developers say it has huge overhead.*
perfmon: *package for Solaris 7, need root privilege to install*
utime: *Linux kernel patch for 2.2.13*

[1] *PAPI requires OSF 5.1 and pfm device driver patch. OSF 5.1 and pfm device driver does not support saving and restoring counters on context switch.*
[2] *PCL does not save counters on context switch.*

Table 3: Software packages for timing applications.

being ported to additional platforms. The developer of PCL has indicated that it is no longer being actively developed. Other options either have limited availability or insufficient resolution. We have developed a standard interface to the timing software packages that is used by our software to hide the details of accessing the individual packages, thus allowing us to easily switch between packages.

Earlier in the year we performed experiments with the low fidelity network representation on the MIPS/IRIX platform (*nirvana.acl.lanl.gov*) mainly because it is the only system that does not require a kernel patch to access the timing information. We tried both the PAPI and PCL timing packages as well as getrusage. (Some preliminary experiments were done with Atom, but not with the application coupled to the DaSSF simulation.)

With the imminent demise of *nirvana*, we have lately focused more attention on trying to use *wolverine*, an ALPHA/LINUX platform, for direct execution experiments involving the medium fidelity network representation. Experiments thus far have used a constant value or getrusage to obtain timing information. PAPI only recently became available for ALPHA/LINUX and we have not yet tested this version.

We encountered several difficulties in doing direct execution of SWEEP3D on *wolverine* with the medium fidelity network model that had not been seen using the low fidelity network model on *nirvana*. SWEEP3D uses Fortran automatic arrays internally. This works fine when SWEEP3D is run separately, but caused problems when the application was coupled to the simulation. We tried unsuccessfully to understand why this problem occurs on *wolverine*, but ultimately worked around it by eliminating the automatic arrays. We encountered a problem in DaSSF with large messages being sent between distributed memory nodes. This problem was resolved with the guidance of the DaSSF developer. We found that the simulation could deadlock when an unexpected message was received on a node. Enhancing the amount of information passed between the replacement MPI library and the simulation corrected this problem.

We are poised to begin direct execution experiments using the higher resolution PAPI timing package. A remaining issue before validated results can be obtained is that time spent in the MPI library is not presently accounted for.

3.3 Complex Statistical Characterization of Workloads

The theoretical and computational issues involved in large-scale network modeling have direct analogues in some of the most challenging and important problems in statistical physics. The complicating features which appear in both contexts include multiple spatial and temporal scales, and strongly correlated dynamics and stochasticity. As a result we are bringing to bear on problems arising in network performance modeling techniques originally developed and already highly successful in the context of nonequilibrium statistical physics.

3.3.1 Rayleigh-Ritz Method & Closure Schemes

One such technique is a numerical Rayleigh-Ritz method [1]. This is a variational formulation for the time dependence of the probability distribution functions of network variables. Whether one uses a discrete event dynamical system or stochastic fluid level approach, the network variables are described by a large system of nonlinear, coupled, stochastic equations. Due to their nonlinear nature, these equations cannot be solved exactly and require that approximations be made. The approximations which describe higher order statistics in terms of lower ones are known as a closure. The Rayleigh-Ritz variational method then tests these statistical closure ideas using the exact dynamics (i.e., the original equations), but more cheaply and quite often under more extreme circumstances than is feasible by direct numerical simulation.

Closure schemes can then be developed by using educated guesses for the system statistics. These "guesses" may be inspired from an analysis of the data using the visualization tool. As a result, empirical data from actual networks and workloads may be exploited in a numerical calculation. Additional statistical information can be calculated within the variational formulation, including multi-time correlations, and the probability of large fluctuations in performance. Moreover, there are internal consistency checks available which may be used as diagnostics to detect *a priori* faulty predictions of the closures, potentially reducing the amount of time spent on inadequate models.

Using this methodology we are attempting to address several types of problems such as the probability of buffer overflow and likelihood of long-time performance averages. Moreover, we expect that well chosen closures will lead to being able to answer these questions with greatly reduced computational effort.

3.3.2 Principal Components Analysis

We have explored our preliminary data sets (see the low fidelity simulation results on page 28) for interesting statistical structure using the method of principal components analysis (PCA). We postprocess our simulation output data to construct a vector $\mathbf{q}(t)$ with components $q_i(t)$ equal to the number of messages waiting to be transmitted through the network from computational node $i = 1 \ldots n$ at time step $t = 0 \ldots T$. We call $\mathbf{q}(t)$ the *queue length* time series. From this we compute a covariance matrix

$$\mathbf{C} = \frac{1}{T} \sum_{t=0}^{T} \left[\mathbf{q}(t) - \overline{\mathbf{q}}\right] \left[\mathbf{q}(t) - \overline{\mathbf{q}}\right]^t , \tag{1}$$

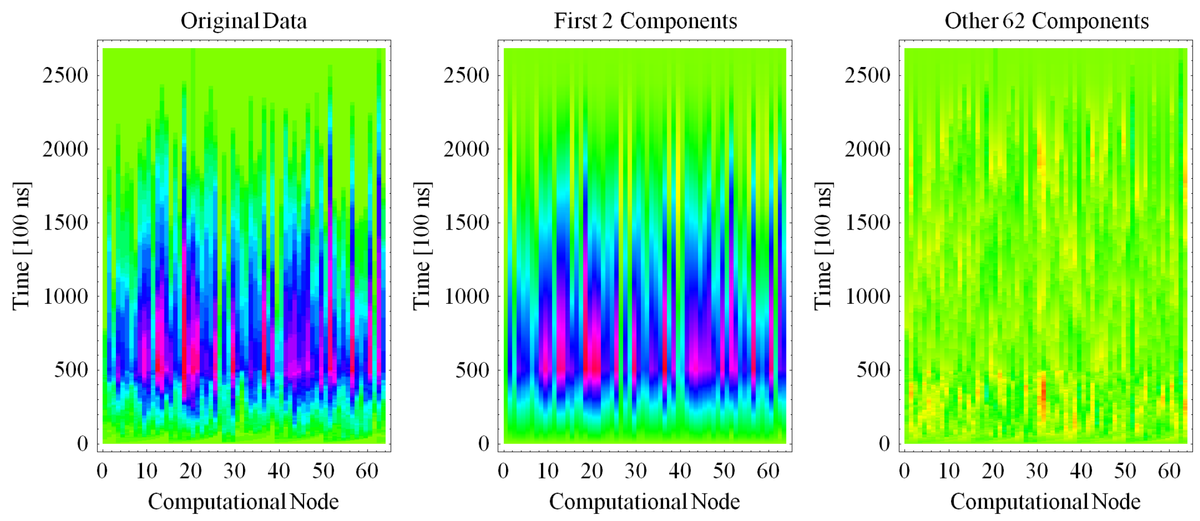

Figure 4: Queue lengths in a simulation with 64 computational nodes (left) and the contribution from the two eigenmodes with statistically significant eigenvalues (middle) and the remaining 62 eigenmodes (right): The horizontal axis represents node number, the vertical axis is simulated time, and colors range from green (queue length of zero) to red (queue length of twelve).

where $\overline{\mathbf{q}} = \frac{1}{T+1}\sum_{t=0}^{T}\mathbf{q}(t)$, and find its eigenvalues λ_i ordered such that $\lambda_i \geq \lambda_{i+1}$. In general, some of the eigenvalues can be attributed to random noise rather than to the correlation structure in the queue length time series. To determine the number of statistically significant eigenvalues, k, we consider random permutations of the original series:

$$q_i'(t) = q_i(t')|_{t'=\mathbb{P}t} \tag{2}$$

where $\mathbb{P}$ is an operator that maps $\{0,\ldots,T\}$ to a random permutation of $\{0,\ldots,T\}$. We can find the largest eigenvalue λ_1' for several realizations of the series $\mathbf{q}'(t)$ to determine the cut-off below which eigenvalues λ_i can be attributed to random noise in the correlation matrix.

Preliminary results indicate that only a few eigenmodes, k, are statistically signification for the "CFD" and "uniform" workloads (see page 28) if the simulated network is operated near its capacity. Figure 4 shows how $k = 2$ modes are sufficient to explain most of the variation in queue length for a simulation with $n = 64$ computational nodes. We hope that further analysis using the workloads generated by real applications on our medium-fidelity network will show similarly encouraging results.

3.3.3 Aggregate Statistical Workload Model for the ASCI SWEEP3D Application

Although our detailed simulation results are based on the direct execution of the SWEEP3D application [38], it is also useful to have a gross model of the application that possesses its basic network-traffic characteristics. The message passing in SWEEP3D can be characterized in terms of its input parameters `npe_i`, `npe_j`, `mk`, `mmi`, `it_g`, `jt_g`, `kt`, and `mm`. Here `npe_i` $\times$ `npe_j` is the number of CPUs used in the partition, which is rectangular.

Messages are passed in the "north-south" and "east-west" directions. The north-south messages have size $8 \times \left\lfloor \frac{\texttt{it\_g}}{\texttt{npe\_i}} \right\rfloor \times \texttt{mk} \times \texttt{mmi}$ while east-west messages have size $8 \times \left\lfloor \frac{\texttt{jt\_g}}{\texttt{npe\_j}} \right\rfloor \times \texttt{mk} \times \texttt{mmi}$, measured in bytes. The total number of each type of message sent per iteration between orthogonally adjacent nodes in the partition is $2 \times \left\lfloor \frac{\texttt{kt}}{\texttt{mk}} \right\rfloor \times \left\lfloor \frac{\texttt{mm}}{\texttt{mmi}} \right\rfloor$. The waiting times between message sends can be crudely approximated by an exponential distribution with a mean of 50 μs.

The workload model described in Section 3.1 allows one to construct precisely such a statistical model of an application like SWEEP3D. One can specify the nodes with which each node communicates, the mean time in the exponential distribution of waiting times between message sends, and the size of the messages.

3.4 Test Workload

To precisely control the content of messages we developed a workload model specifically for testing. In this workload model, the user specifies exactly what messages will be sent from each SMP node in the network topology, the time the message is sent, the destination of the message, the message size in bytes, and optionally, the data content of the message.

Table 4 summarizes the test cases that currently comprise the regression testing suite for the medium fidelity network. The information includes the number of switches present in the network topology, the number of messages sent, the number of packets sent, when the AckNow request is sent, the source and target nodes of the message, the time when the message is sent, whether a data payload is included in the message, and notes that briefly characterize the purpose of the test. This suite allows us to verify the network behavior in relatively straightforward situations (testA–testH), as well as in more complicated situations where we force the use of both virtual channels and create contention for channel use (testJ–testQ). The remaining tests verify a simplified Quadrics routing protocol (testR) and a more accurate protocol (testS) which includes wildcard routing.

3.5 Ping Workload

The ping workload representation was developed to facilitate comparison of the simulation with ping tests conducted on the network hardware. Parameters for this workload are the exact size of the message, the exact delay between messages, the number of messages to send, and the message destination for each message source.

4 Describing and Constructing Hardware Architectures

4.1 Connectivity of Fat-Tree Networks

For simplicity we consider so-called "single-rail" networks where each computation node only has one network interface card (NIC). We also assume each network switch has eight duplex I/O ports; the ports may be linked to computational nodes or to other switches. The network is organized into layers of switches that connect only to the layers above and below: for eight-port switches, we have four upward connections and four downward connections. Thus we have a quaternary fat-tree network: "quaternary" because each switch has four-fold connections upwards and downwards, and "fat" because the number of switches per layer is the same for all layers. (Note that the real-life networks are typically "thinned" at the higher layers be reducing the number of switches there.) Figure 5 illustrates the layout of such a network with 64 computational nodes and three layers of 16 switches each.

We start with some formalism describing these networks. Let L be the number of layers in a complete (i.e., not thinned) quaternary fat-tree network. Number the layers $\ell = 1, \ldots, L$. Each layer has 4^{L-1} switches, so we label these with IDs $x = 0, \ldots, 4^{L-1} - 1$. This means there are

test	switches	messages	packets	ack	source	target	time	data	notes
testA	1	1	1	64	0	1	10	no	
testB	1	1	1	end	2	1	10	no	
testC	1	1	few	64	0	2	10	yes	
testD	1	1	few	end	1	0	10	null	
testG	2	1	1	64	4	3	10	no	2 switches
testH	2	1	1	end	0	2	10	yes	2 switches
testJ	1	2	1	64	0	1	10	no	2 channels used
			1	end	2	1	10	no	
testK	1	2	1	64	0	1	10	no	2 channels used
			1	64	2	1	10	no	
testL	1	2	1	end	0	1	10	no	2 channels used
			1	end	2	1	10	no	
testM	1	3	1	64	0	1	10	no	channel contention
			1	64	2	1	10	no	
			1	64	7	1	10	no	
testN	1	5	1	64	0	1	10	no	channel contention
			1	64	2	1	10	no	fairness
			1	64	4	1	10	no	
			1	64	7	1	310	no	
			1	64	3	1	510	no	
testO	1	3	1	64	0	1	10	no	overlapping and
			1	64	2	1	200	no	skewed packets
			1	64	7	1	700	no	
testQ	1	2	1	64	0	1	10	yes	crossing messages
			1	64	1	0	110	yes	
testR	8	1	1	end	0	13	10	no	Quadrics1 routing
testS	8	1	1	end	0	13	10	no	Quadrics2 routing

Table 4: Medium fidelity network test cases.

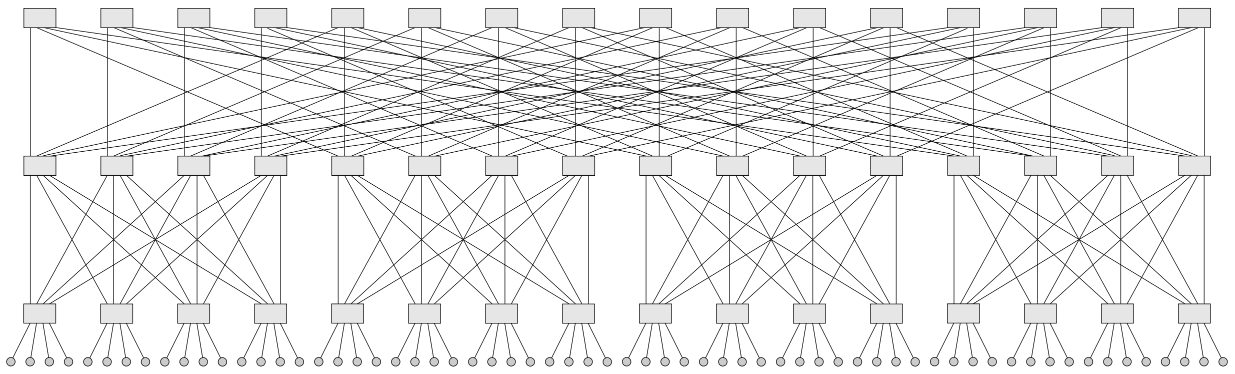

Figure 5: *Representation QS:* Quaternary fat-tree network with 64 computational nodes (small circles along the bottom) and three layers of 16 switches each (rectangles). Each switch has four duplex connections to the layer above it and four duplex connections to the layer below it—the lines in the diagram represent these connections.

$s_L = L \cdot 4^{L-1}$ switches in the network and $n_L = 4^L$ nodes connected to it. Now consider the port p leading up from a switch labeled x in layer ℓ into port $q = q(x, p, \ell)$ of a switch labeled $y = y(x, p, \ell)$. It is useful to express these switch labels in base four: let $x = \sum_{n=1}^{L-1} 4^{n-1} x_n$.

An "ideal" quaternary fat tree obeys the simple wiring rule

$$y(x, p, \ell) = 4^\ell \left\lfloor \frac{x}{4^\ell} \right\rfloor + 4^{\ell-1} p + \left(x \bmod 4^{\ell-1} \right). \tag{3}$$

We want to consider a real Quadrics network, however, that is not organized in this ideal manner. The network has six levels, but the connections between levels 5 and 6 are thinned by 75%. The deviation from the ideal situation results from constraints due to hardware components, chassis, cabling, and siting.

The first two layers are connected via QM401 cards containing two rows of four switches [6, 5]:

$$q(x, p, 1) = x_1, \tag{4}$$
$$y_1(x, p, 1) = p, \tag{5}$$
$$y_k(x, p, 1) = x_k \text{ for } k \geq 2. \tag{6}$$

Layer two is connected to layer three where the upper ports of QM401 cards are connected to the lower ports of QM402 cards [6, 5]:

$$q(x, p, 2) = x_2, \tag{7}$$
$$y_1(x, p, 2) = p, \tag{8}$$
$$y_2(x, p, 2) = x_1, \tag{9}$$
$$y_k(x, p, 2) = x_k \text{ for } k \geq 3. \tag{10}$$

Layer three is connected to layer four where the upper ports of QM402 cards are connected through QM407 cards and then cables to the lower ports of QM401 cards in the mid-level switches [6, 5]:

$$q(x, p, 3) = x_3, \tag{11}$$
$$y_1(x, p, 3) = x_4, \tag{12}$$
$$y_2(x, p, 3) = p, \tag{13}$$
$$y_3(x, p, 3) = x_1, \tag{14}$$
$$y_4(x, p, 3) = x_2, \tag{15}$$
$$y_k(x, p, 3) = x_k \text{ for } k \geq 5. \tag{16}$$

Layer four is connected to layer five via QM401 cards containing two rows of four switches [6, 5]:

$$q(x, p, 4) = x_1, \tag{17}$$
$$y_1(x, p, 4) = p, \tag{18}$$
$$y_k(x, p, 4) = x_k \text{ for } k \geq 2. \tag{19}$$

Layer five is connected to layer six where the upper ports of QM401 cards are connected through QM412 cards, then QM407 cards, and then cables to the lower ports of QM401 cards in the top-level switches [6, 5]:

$$q(x, 0, 5) = x_5, \tag{20}$$
$$y_k(x, 0, 5) = x_k \text{ for } k \leq 4, \tag{21}$$
$$y_5(x, 0, 5) = 0. \tag{22}$$

Note that ports $p = 1, 2, 3$ are not used.

All of the above equations can be written succinctly as

$$\tilde{y}_i = \tilde{x}_{w_{\ell,i+1}} \text{ for } i = 0, \ldots, L, \tag{23}$$

where $\tilde{y} = [q, y_1, \ldots, y_{L-1}]$, $\tilde{x} = [p, x_1, \ldots, x_{L-1}]$, and

$$\mathbf{w} = \begin{bmatrix} 1 & 0 & 2 & 3 & 4 & 5 \\ 2 & 0 & 1 & 3 & 4 & 5 \\ 3 & 4 & 0 & 1 & 2 & 5 \\ 1 & 0 & 2 & 3 & 4 & 5 \\ 5 & 1 & 2 & 3 & 4 & 0 \end{bmatrix}. \tag{24}$$

Note that an ideal fat tree (Eq. 3) becomes

$$\mathbf{w} = \begin{bmatrix} 1 & 0 & 2 & 3 & 4 & 5 \\ 2 & 1 & 0 & 3 & 4 & 5 \\ 3 & 1 & 2 & 0 & 4 & 5 \\ 4 & 1 & 2 & 3 & 0 & 5 \\ 5 & 1 & 2 & 3 & 4 & 0 \end{bmatrix} \tag{25}$$

in this notation. One can write the inverse of Equation 23 as

$$\tilde{x}_i = \tilde{y}_{v_{\ell,i+1}}, \tag{26}$$

where

$$v_{\ell,i+1} = j \text{ such that } w_{\ell,j+1} = i \tag{27}$$

so

$$v_{\ell,w_{\ell,i+1}+1} = i. \tag{28}$$

We now consider an upward path of ports $p^{(\ell)}$ that passes through switches $x^{(\ell)}$, for $\ell = 1, \ldots, L$. We write $\tilde{x}^{(\ell)} = [p^{(\ell)}, x_1^{(\ell)}, \ldots, x_{L-1}^{(\ell)}]$ and $\tilde{x}'^{(\ell)} = [q^{(\ell)}, x_1^{(\ell)}, \ldots, x_{L-1}^{(\ell)}]$, so the path can be expressed as

$$x_i'^{(\ell+1)} = \tilde{x}^{(\ell)}_{w_{\ell,i+1}} \equiv \tilde{x}^{(\ell)}_{f_\ell(i)}, \tag{29}$$

where $q^{(\ell+1)}$ is the destination port from level ℓ to level $\ell + 1$, and the function $f_\ell(i) \equiv w_{\ell,i+1}$ is used for notational convenience. Since the links are duplex, the $q^{(\ell)}$ represent a downward path through the same switches, and

$$q^{(\ell+1)} = \tilde{x}^{(\ell)}_{f_\ell(0)} = x^{(\ell)}_{f_\ell(0)} = \tilde{x}'^{(\ell)}_{f_\ell(0)}, \tag{30}$$

provided $w_{\ell,1} \neq 0$. We can apply Equations 29 and 30 recursively to obtain the closed form expression:

$$\begin{aligned} q^{(2)} &= x^{(1)}_{f_1(0)}, \\ q^{(3)} &= x^{(2)}_{f_1(f_2(0))}, \\ &\ldots \\ q^{(\ell+1)} &= x^{(\ell)}_{f_1 \circ \ldots \circ f_\ell(0)}, \end{aligned} \tag{31}$$

provided $f_1 \circ \ldots \circ f_\ell(0) \neq 0$. If the $q^{(2)}, \ldots, q^{(L)}$ are a permutation of $x_1, \ldots, x_{L-1}$, then the connection matrix $\mathbf{w}$ specifies a network where the downward path uniquely determines the level-one destination switch for the path. For the real Quadrics matrix (Eq. 24) and for the idealized fat-tree matrix (Eq. 25) we have the particularly simple relationship $q^{(i+1)} = x_i^{(1)}$.

4.2 A Hardware Description Language

Machines are modeled as constructions of multiple instances of parameterized components–top-level entities–defined as (derived from) SSF entities. Currently the components we have defined include SMP nodes, NICs, and network switches; this set may be expanded to include WAN components such as routers. By *top level* we mean entities that it makes sense to regard as the basic building blocks of a complete model; that each such entity might itself comprise some number of more primitive entities (such as the various components of an SMP node, e.g. CPU, in turn comprising processing core, various caches and buses, etc.) is abstracted.

Given the definitions of a set of components (normally compiled into a software library) the next level of organization is their instantiation and assembly—specification of the multiplicity, actual parameterization, and interconnection to yield a particular model. As stated a model may be defined programmatically in the main function and so effectively 'hard-wired,' or specified in DML. The use of DML is the higher-level and more flexible and abstract approach; additionally it defers model construction from compile time to run time (model simulation time). For these reasons we use the latter approach.

For large models—that is, those comprising a large number of components—even specification in DML quickly becomes cumbersome, running to millions of lines of text for the models in which we are interested. This motivated the development of a tool to provide yet another level of abstraction. The obvious approach is to write a program that generates DML for some space of models, and this was used as an interim solution until a more general tool was developed that translates a yet higher-level (than DML) language into DML.

The invariant tension in programming language design is that between specificity, such as that provided by assembly languages, and various forms of abstraction (from the hardware, in supporting higher-level programming paradigms), as provided by so-called higher-level languages such as C++. The design of C++ exhibits a useful approach to reducing this tension: C++ is a superset of C—a somewhat lower level language—which in turn can incorporate assembly language in-line. Using a roughly analogous approach we have defined a machine representation language suited to very compact representation of particular topological structures in which we are interested, e.g. quaternary fat trees, and with special recognition of the components of concern, while preserving the ability to write directly in DML. This is a light-weight language implementation in the sense that it required only one month to design and implement.

Details of the design and use of this hardware description language are beyond the scope of this report; the complete description is available elsewhere [23]. Instead, a single example serves to give a sense of the economy of the notation.

```
l0=n[4096]
l1=s[1024]
l2=s[1024]
l3=s[1024]
l4=s[1024]
l5=s[1024]
l6=s[1024]

connect l0,l1
connect l1,l2 [1, 0, 2, 3, 4, 5]
connect l2,l3 [2, 0, 1, 3, 4, 5]
connect l3,l4 [3, 4, 0, 1, 2, 5]
connect l4,l5 [1, 0, 2, 3, 4, 5]
connect l5,l6 [5, 1, 2, 3, 4, 0]

model.dml{}
```

expands to approximately 6MB DML ASCII text and fully specifies a model of a 4096-node machine arranged as a six-layer quaternary fat tree, a mathematical description of which may be found elsewhere [8].

5 Low-Fidelity Quadrics Network Simulation Design and Implementation

The *à la carte* low-fidelity network implementation is derived from the prototype models we used to initially evaluate DaSSF. It represents the network as a circuit-switched fat-tree network. The source-target patterns for messages can be configured. The switches with four *up* ports and four *down* ports are modeled at the packet level with a simple protocol that allocates a circuit through the network for each message. Using a single circuit for an entire message may be adequate for some types of simulation studies where the details of network traffic congestion are not important. We have run the low-fidelity simulation with a simple statistical workload on a variety of sample models from 1 to 4096 computational nodes (up to 6144 switches).

5.1 Requirements

The requirements for the evaluation prototype and hence the low-fidelity network representation were described in Section 2.

5.2 Design

We used the results of our domain analysis to define a subset of components to implement in the implementation. We chose not to model the processors and memory hierarchy of an SMP node in any detail and instead to focus on modeling the interconnection network between nodes as a fat-tree network with a circuit-switched routing protocol. For the workload, rather than model the message traffic from any specific application, we chose to have each processor node emit messages with exponentially distributed interarrival times. The message destination node is selected with uniform probability, and the message size is exponentially distributed. The parameters for the distributions are inputs to the simulation.

The DaSSF API provides five base classes that applications may subclass. The **SSF_Entity** class defines the entities in the simulation. The **SSF_Process** class defines the behaviors that entities possess. Entities are connected to each other via channels, an **SSF_OutChannel** in the transmitting entity and an **SSF_InChannel** in the receiving entity. An **SSF_Event** represents the information that flows between entities across the channels. Parameterized properties of model components, the number to be instantiated, and their connectivity are specified via an input file written in the Domain Modeling Language (DML). A simplified UML (Unified Modeling Language) diagram of our low-fidelity model is shown in Figure 6. The model contains three types of entities, representing the SMP node, the network interface card (NIC), and the network switch.

The SMP node has an outgoing channel for sending messages to its NIC and an incoming channel for receiving messages from its NIC. The NIC has a corresponding incoming channel for receiving messages from its SMP node and an outgoing channel for sending messages to its SMP node. Additionally, the NIC has an outgoing channel and an incoming channel that connect it to its network switch. Each network switch has 8 incoming channels and 8 outgoing channels. At the level nearest the NICs, 4 of the channels communicate with 4 NICs and 4 communicate with the next level in the fat tree. (An example of a fat-tree network is shown in Figure 5). Higher in the fat tree communication involves only other switches. The route that a message takes through the

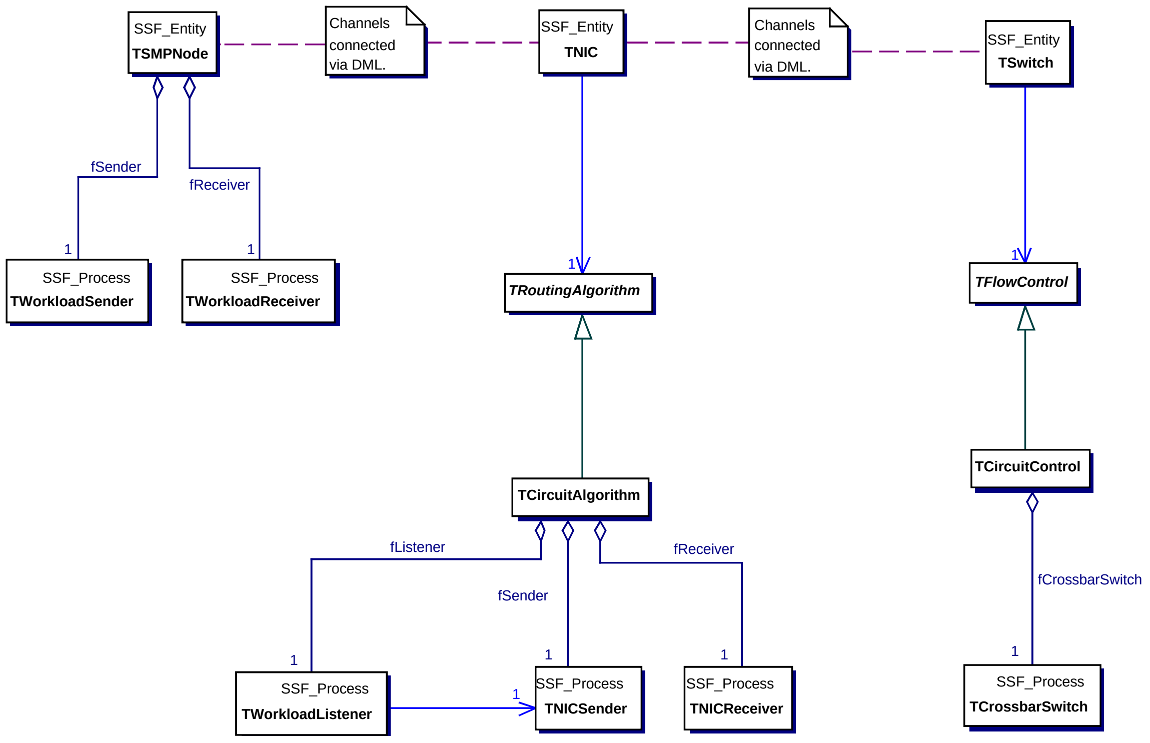

Figure 6: Simplified UML (Unified Modeling Language) class diagram for the low-fidelity simulation.

network is determined at the source and is the same for all packets in a message. The route may be read and stored in a routing table or computed via a routing method.

The SMP node has two processes, **TWorkloadSender** for sending messages and **TWorkloadReceiver** for receiving messages. The NIC has a routing algorithm which has three processes, **TWorkloadListener** for receiving messages from the source SMP and buffering them if the NIC is busy, **TNICSender** for splitting the message into packets and sending the packets to the network switch, and **TNICReceiver** for receiving packets from the switch, reassembling the message and sending it to the destination SMP. The network switch has a flow control algorithm which has one process, **TCrossbarSwitch**, that receives packets on an incoming channel from a NIC or another switch and forwards them out the appropriate outgoing channel to the next switch or NIC as specified by the route that is embedded in the packet.

We defined separate processes for the actions performed by the SMP node and NIC entities as a way of modularizing the logic, to reflect the fact that a NIC may represent a significant process running asynchronously with the SMP, and to allow for the possibility of multiple NICs per SMP. However this necessitates the use of other mechanisms such as a semaphore or an internal channel to allow processes belonging the same entity to communicate with each other. Rather than attaching the processes for message and packet handling directly to the NIC, we interposed the abstract **TRoutingAlgorithm** object and created a **TCircuitAlgorithm** class that contains the processes which implement a circuit routing algorithm for packet delivery. This gives us the flexibility to define other algorithms in the future and give new behavior to the NIC by simply selecting at runtime a different algorithm to be constructed. Similar logic pervades the design of the switch and its process.

Two types of events are defined, one for a message, and one for the packets in a message. Packets are further subclassed into four types: the open circuit packet, the acknowledge circuit packet, the close circuit packet, and the data packet. The precise behavior that occurs in the processes depends upon the type of packet that arrives.

Our design employs the DaSSF-supplied random number generation module. We collect traces of packet movement through the network using DaSSF's data packing and dumping capability.

5.3 Implementation

The present implementation of our low-fidelity simulator consists of approximately five thousand lines of ANSI- and POSIX-compliant C++ that runs using MPI under the Linux, Solaris, and Irix operating systems. We use DaSSF's built-in user threads to avoid the operating-system overhead associated with creating hundreds of native threads. We have executed the code on single processor boxes, SMPs, and clusters of workstations on a LAN. Since the specifications of modelled components and architectures are entirely independent of the host platform, models may be developed and executed on any convenient or appropriate host.

The data-collection facility in the implementation permits one to collect detailed information concerning the history of each simulated message: its movement from the computational node to the network interface card, the opening of a network circuit for its transmission, its packetization, its acknowledgement, and the closing of its network circuit. The implementation has been tested to verify correctness of time delays, probabilistic distributions, and synchronization behavior. The data-collection also supports our visualization capability, allowing users to view message traffic in the simulated system. Data summaries supply information on queue sizes, throughputs, port usage, timeouts, path lengths, and communication patterns.

5.4 Results

We have constructed a variety of architecture models for testing the simulation and for measuring its performance and scalability. The simplest models consist of computational nodes connected to each other by a bus or via a pair of network interface cards; slightly more complex models contain one or two network switches with eight computational nodes. These small models provide a testbed for verifying that the timing delays in various parts of the simulation match those of the hardware being simulated. They also supply a convenient platform for debugging and detailed tracing of the simulation's progress. Even a simulation of a small system executing for a short time can produce a large amount of data if the history of each simulated entity, process, and event is recorded.

Our larger models contain 64, 128, . . . , up to 4096 computational nodes networked together in a fat-tree topology. Figure 5 shows the layout of a 64-node system. A 4096-node system (similar in size to a proposed ASCI Q machine architecture) uses these 64-node systems as building blocks—the nodes of one such system are each replaced with a 64-node system. Having these more realistically-sized models allows us to undertake meaningful studies of system performance and scaling—both of the simulation system itself and of the architectures being simulated. Our current studies focus on the behavior of the simulation system rather than the simulated architecture. In Section 5.5 we discuss the scaling properties of this simulation.

Figure 7 shows the results of a simulation of the 64-node, 48-switch architecture shown in Figure 5, with a load of 100 MB/s of message traffic originating at each node. The diagram shows that many of the messages reach their destination in the minimum possible time (based on network connectivity and time delays), but that a significant fraction of the messages have additional delays due to network congestion. The mean message size is 4 KB. The simulation is not realistic in the sense that the network protocols do not correspond to those of any actual network hardware and

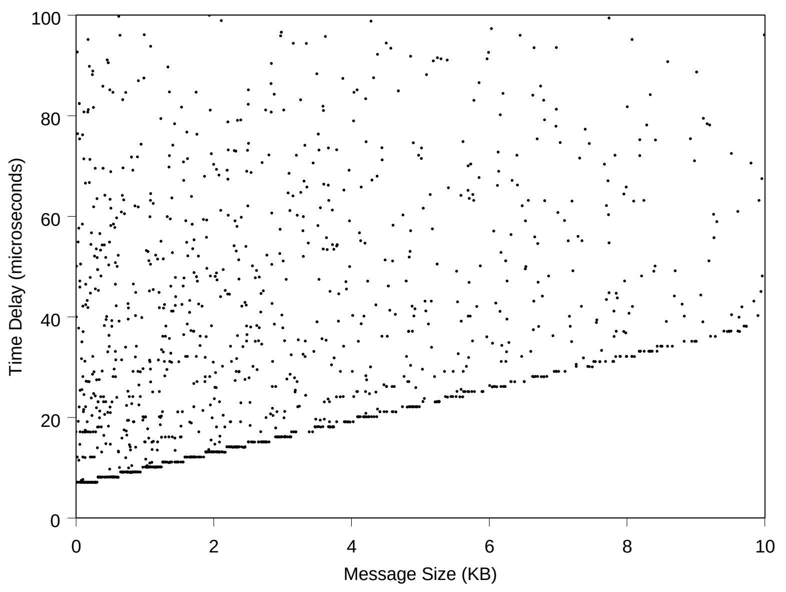

Figure 7: Example output data from a 64-node simulation where each node generates an average of 100 MB/s of message traffic on the network. The mean message size is 4 KB.

the workload is highly generalized. Nevertheless, the simulation does show typical characteristics for a moderately-loaded system.

We have also used the simulation to compare two typical types of application communication patterns. Our "uniform" pattern represents applications that send messages at random between computational nodes: specifically, each computational node has equal probabilities of sending a message to any other computational node. Figure 8 illustrates the sort of network traffic such a workload creates in our simulation. Our "CFD" pattern represents applications that send messages to neighbors on a computational grid, such as a computational fluid dynamics application does: specifically, the computational nodes are assigned locations on a three dimensional cube (e.g., on a $4 \times 4 \times 4$ cube for a 64-node simulation or on a $16 \times 16 \times 16$ cube for a 4096-node simulation), and each node has equal probabilities of sending a message only to its immediate neighbors on the grid. Figure 9 illustrates that sort of network traffic in our simulation. The CFD pattern clearly uses the higher layers of the fat tree much less than the uniform pattern.

5.5 Scaling Behavior

We are able to simulate 4096-node clusters using computing platforms such as the SGI Origin 2000; peak memory usage was about 8 GB in our initial prototype. Table 5 shows some of the performance figures we have obtained for a 64-node cluster. Note that performance degrades as the number of computational nodes increases because the processing nodes are not performing computational work; essentially only message passing is taking place. Simulations involving the direct execution of actual applications will have a high proportion of their CPU time spent on the workload representation, and hence will exhibit more favorable scaling characteristics.

We have also investigated how the simulation scales as a function of problem size. Figure 10 illustrates the fairly linear scaling in terms of compute time and peak memory usage for simulations of fat-trees with 8, 64, 216, 512, and 1000 computational nodes. Since the simulations were all run on a single CPU, the scaling behavior does not include contributions from additional communications overhead as would be the case if multiple CPUs were used for the larger cases. Note that the

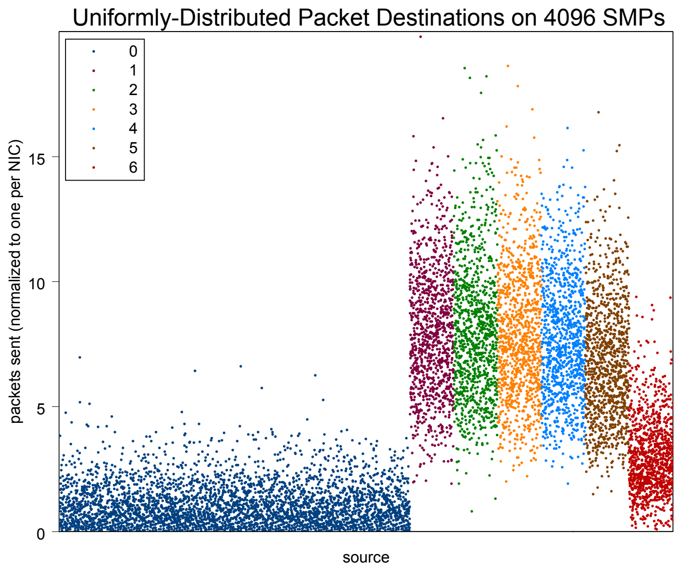

Figure 8: Number of message packets for the "uniform" pattern of workload communication in a 4096-node simulation: The horizontal axis represents the simulation entity (nodes in dark blue, and switches in other colors for each level) and the vertical axis represents the number of packets sent by the entity, normalized on a scale so that the nodes send an average of one packet.

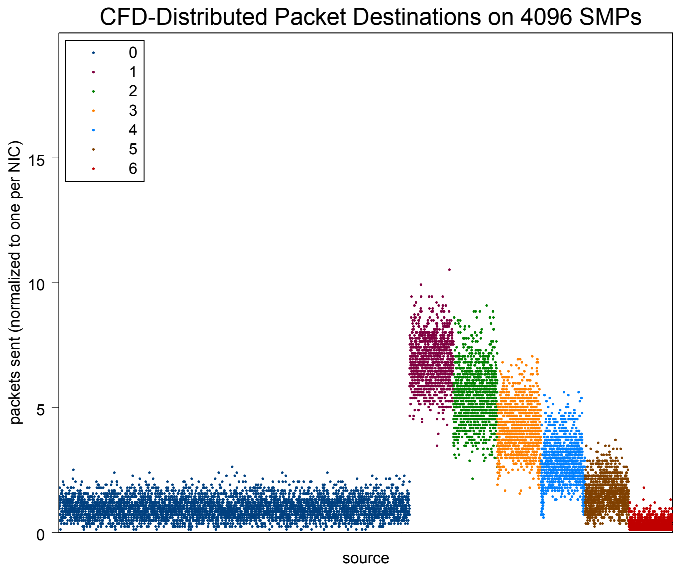

Figure 9: Number of message packets for the "CFD" pattern of workload communication in a 4096-node simulation: The horizontal axis represents the simulation entity (nodes in dark blue, and switches in other colors for each level) and the vertical axis represents the number of packets sent by the entity, normalized on a scale so that the nodes send an average of one packet.

Platform	Computational Nodes	Events (1/sec)	Execution Time (sec)
Linux, 733 MHz Pentium III	1	2020.2	178.1
	2	689.9	521.6
Linux, 500 MHz Pentium III	4	494.3	728.0
	8	379.9	947.3
Solaris, Sun SPARC Ultra 5	1	1105.1	325.6
Irix, Origin 2000	1	1298.3	277.2
	4	1351.1	266.4
	16	630.2	571.1

Table 5: Performance of the initial prototype simulating at 64-node architecture on a variety of computing platforms.

"CFD" and "uniform" workloads have similar run times, but the latter uses more memory because messages tend to queue up at the network interface cards since the network bandwidth is not sufficient to handle the demand.

The memory usage measurements discussed so far in this section were made using DaSSF prior to its version 3.2. For version 3.2, memory allocators were rewritten to avoid fragmentation. Preliminary indications are that the simulation runs described above would require only about half of the memory if version 3.2 were used. We have also noticed that the DaSSF model partitioner tends to use disproportionately large amounts of memory if presented with a large DML input file: a DML input file containing the routing table for a 4096-node fat tree required about 10 GB of memory to partition the model.

6 Medium-Fidelity Quadrics Network Simulation Design, Implementation, and Validation

The *à la carte* medium-fidelity network model builds on the low-fidelity model discussed in the previous section by enhancing its accuracy and realism. This model is much closer to representing actual hardware and mimicking the behavior of network protocols in use on real systems. We expect the resolution of this representation to be sufficient for even the most detailed network studies.

6.1 Requirements

Our primary requirement is the ability to accurately model the movement of packets in Quadrics networks consisting of Elan network interface cards [32, 33, 34] connected to Elite switches [35] in a fat-tree network at nearly *flit* (16-bit units) resolution. Figure 11 illustrates the sequence of operations it takes to move a message from one computational node to another through this sort of network. We need to accurately track the movement of the message across the PCI bus between main memory and the NIC, account for its packetization, and clock the transfer of data across the network. Because contention may exist in the network, different parts of the packet may move at different speeds through the switches (i.e., buffering and delays may occur anywhere in the network).

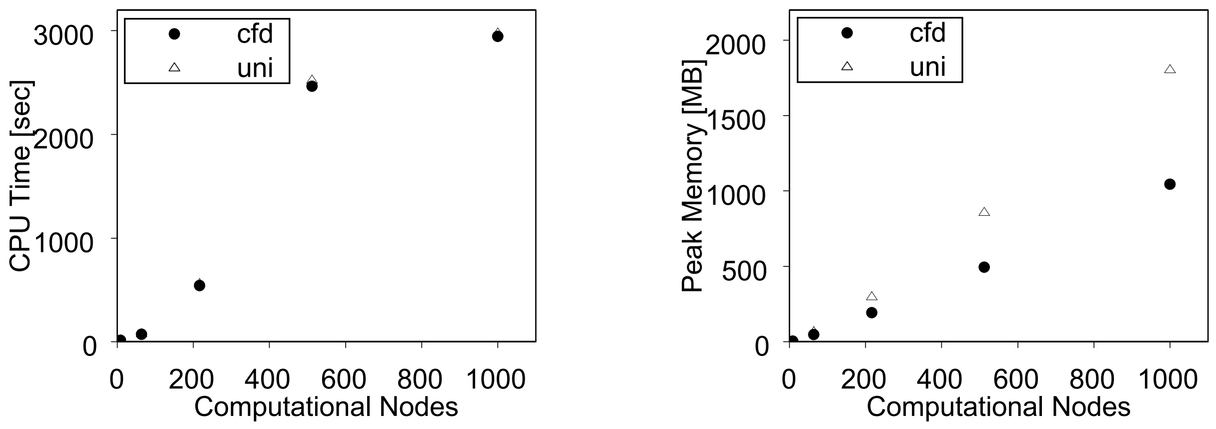

Figure 10: Scaling behavior of a fat-tree simulation on single 1 GHz Pentium III hardware running Linux: the horizontal axes are the number of computational nodes in the simulation and the vertical axes are (left) the amount of CPU time required to complete a 50 microsecond simulation with heavy message traffic and (right) the peak memory used by the simulation. The filled-in circles show results for a "CFD" workload and the hollow triangles show results for a "uniform" workload (see page 28).

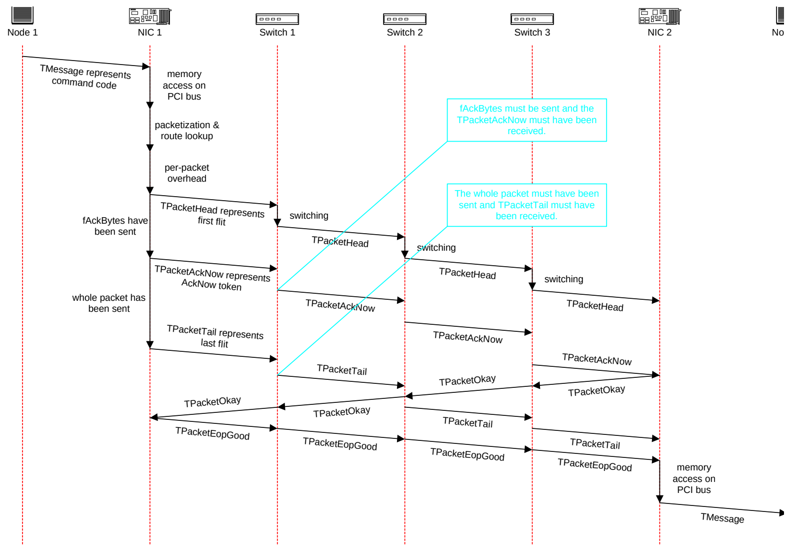

Figure 11: Steps in the movement of a message from one computational node to another in a Quadrics network of Elan network interface cards and Elite switches.

In addition to more accurate representation of network, we also need additional tools to assemble networks that are not perfect fat-trees. This includes linking several fat trees together, and handling trees that are thinned (i.e., have fewer switches) in their upper layers. It is also necessary to have the ability to input cable lengths, as these range over orders of magnitude (from centimeters to meters) in real networks.

6.2 Design

It is a fairly simple matter to augment the design for our low-fidelity network representation to the higher fidelity required. The basic **TSMPNode**, **TNIC**, and **TSwitch** entities remain essentially unchanged. We use object factory methods to generate the appropriate **TRoutingAlgorithm** or **TFlow-Control** instances for the fidelity specified in the DML input file. The medium fidelity **TCircuitAlgorithm** and **TCircuitControl** classes are similar to their low fidelity counterparts except for their internal logic, which is considerably more complex, and the type of packets they handle. The design relies on the tracking of the head and tail of the packet throughout its history, along with various flit-level tokens specified in the Elan protocol. Figure 12 shows the class diagram for the medium-fidelity simulation.

In addition to tracking the movement of the head and the tail of packets through the NICs and switches, we account for the existence of two virtual channels sharing bandwidth at switches (but without age-based priorities, etc.). The Elan *AckNow* request may occur anywhere within the packet (usually after 64 bytes or at the end of the packet). The *EOP_GOOD* tokens free the virtual channels used by the packet. The *STARTx/STOPx* tokens are accounted for by buffering of packets at the incoming links to switches if no outgoing virtual channel is available. We model the PCI bus as half-duplex, and account for writes to the NIC's command port. Finally, we allow wildcards for packet routing on upward links. We do not yet model error conditions. We believe this is nearly the highest fidelity we can achieve without simulating at the flit level, which would be prohibitively slow. (Adding further resolution may not be cost effective because of the uncertainties involved in the performance of the operating system on the node, memory issues, and PCI bus behavior.)

The basic strategy for dealing with packets is as follows: When the head of the packet reaches an entity like a NIC, switch, or node, it leaves a **TPacketReservation** entity at the entity. The head of the packet is forwarded along the route as soon as possible—it might be delayed slightly for switch logic or may be delayed significantly if it is queued for later transmission. As soon as the head leaves the entity, the reservation starts keeping track of how many bytes remain to be transmitted. The tail of the packet cannot be forwarded until it has been received from the previous stage and the number of bytes remaining is zero. The "okay" event proceeds along the reverse path at full speed, and the "good" event cleans up the reservations. There is some fairly complex timekeeping logic for multiplexing the transmission of packets in switches and the receipt of them in NICs.

6.3 Implementation

We track the leading edge ("head"), trailing edge ("tail"), *AckNow* token, *PACK_OK* token, and *EOP_GOOD* token for packets in the network, which are implemented as **TPacketHead**, **TPacketTail**, **TPacketAckNow**, **TPacketOkay**, and **TPacketEopGood** subclasses of our existing **TPacket** class (see Fig. 12). The **TPacket** class differs from its low-fidelity counterpart in that it supports virtual channels and is tailored for its subclasses that track flit-level events. In Figure 13 we outline the following scenario for the transmission of a message from one computational node to another.

1. A **TWorkloadSender** process creates a **TMessage** event and writes it to the **TSMPNode.fOutBus** channel. This represents the command code being written from main memory.

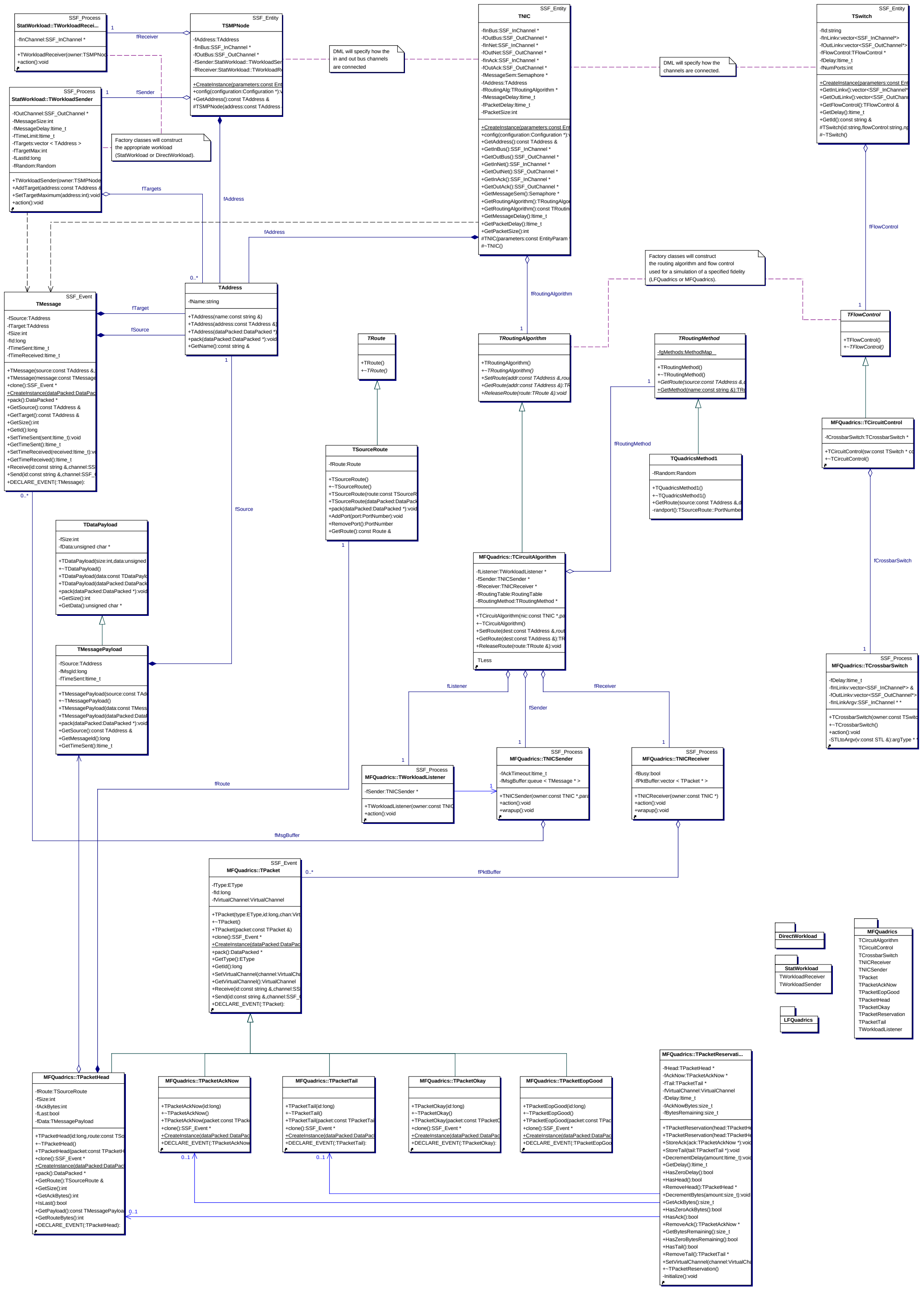

Figure 12: UML class diagram for the medium-fidelity network representation.

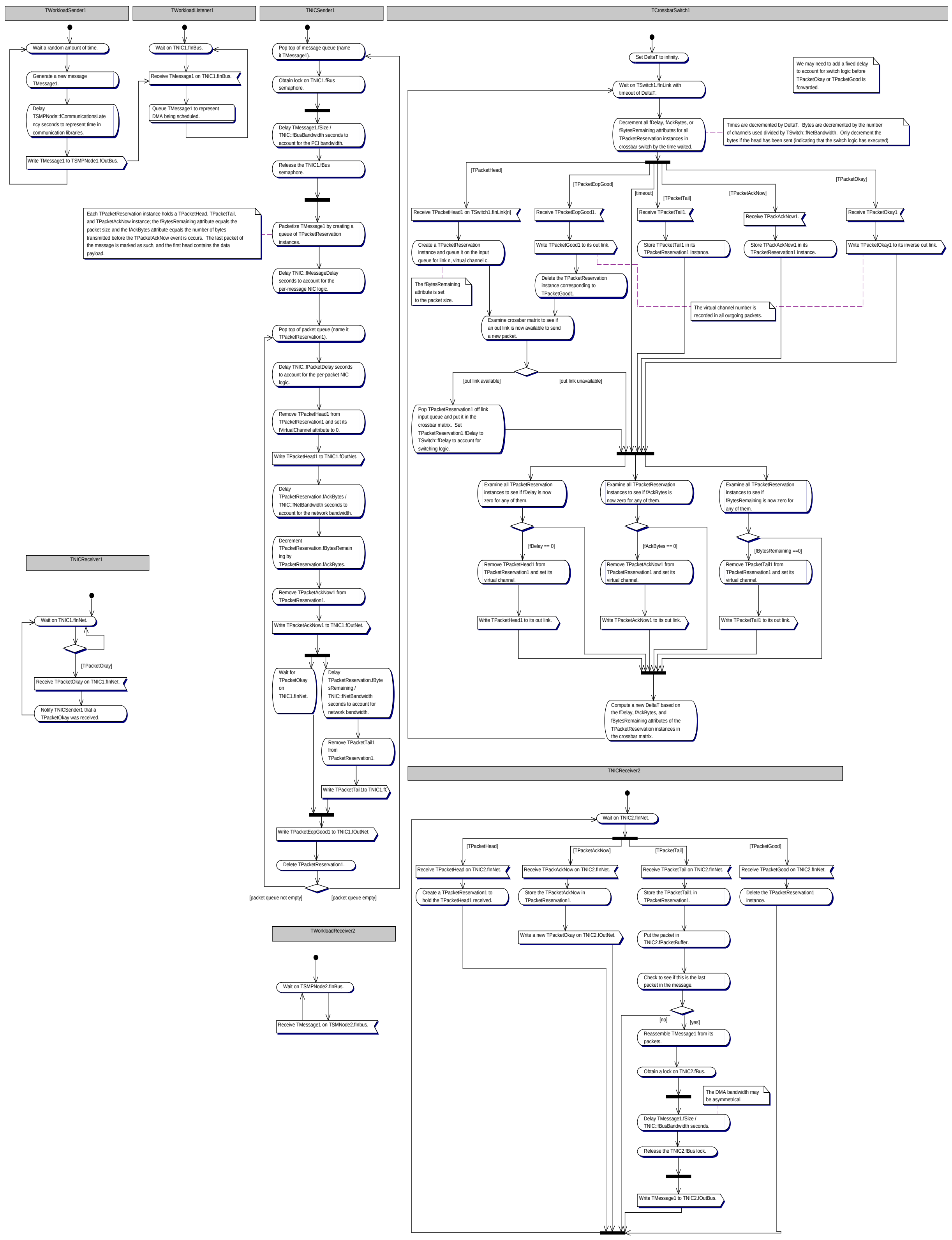
TWorkloadSender1
TWorkloadListener1
TNICSender1
TCrossbarSwitch1
Wait a random amount of time.
Generate a new message TMessage1.
Delay TSMPNode::fCommunicationsLatency seconds to represent time in communication libraries.
Write TMessage1 to TSMPNode1.fOutBus.
Wait on TNIC1.fInBus.
Receive TMessage1 on TNIC1.fInBus.
Queue TMessage1 to represent DMA being scheduled.
Pop top of message queue (name it TMessage1).
Obtain lock on TNIC1.fBus semaphore.
Delay TMessage1.fSize / TNIC::fBusBandwidth seconds to account for the PCI bandwidth.
Release the TNIC1.fBus semaphore.
Each TPacketReservation instance holds a TPacketHead, TPacketTail, and TPacketAckNow instance; the fBytesRemaining attribute equals the packet size and the fAckBytes attribute equals the number of bytes transmitted before the TPacketAckNow event is occurs. The last packet of the message is marked as such, and the first head contains the data payload.
Packetize TMessage1 by creating a queue of TPacketReservation instances.
Delay TNIC::fMessageDelay seconds to account for the per-message NIC logic.
Pop top of packet queue (name it TPacketReservation1).
Delay TNIC::fPacketDelay seconds to account for the per-packet NIC logic.
Remove TPacketHead1 from TPacketReservation1 and set its fVirtualChannel attribute to 0.
Write TPacketHead1 to TNIC1.fOutNet.
Delay TPacketReservation.fAckBytes / TNIC::fNetBandwidth seconds to account for the network bandwidth.
Decrement TPacketReservation.fBytesRemaining by TPacketReservation.fAckBytes.
Remove TPacketAckNow1 from TPacketReservation1.
Write TPacketAckNow1 to TNIC1.fOutNet.
Wait for TPacketOkay on TNIC1.fInNet.
Delay TPacketReservation.fBytesRemaining / TNIC::fNetBandwidth seconds to account for network bandwidth.
Remove TPacketTail1 from TPacketReservation1.
Write TPacketTail1to TNIC1.fOutNet.
Write TPacketEopGood1 to TNIC1.fOutNet.
Delete TPacketReservation1.
[packet queue not empty]
[packet queue empty]
TNICReceiver1
Wait on TNIC1.fInNet.
[TPacketOkay]
Receive TPacketOkay on TNIC1.fInNet.
Notify TNICSender1 that a TPacketOkay was received.
TWorkloadReceiver2
Wait on TSMPNode2.fInBus.
Receive TMessage1 on TSMNode2.fInBus.
Set DeltaT to infinity.
We may need to add a fixed delay to account for switch logic before TPacketOkay or TPacketGood is forwarded.
Wait on TSwitch1.fInLink with timeout of DeltaT.
Decrement all fDelay, fAckBytes, or fBytesRemaining attributes for all TPacketReservation instances in crossbar switch by the time waited.
Times are decremented by DeltaT. Bytes are decremented by the number of channels used divided by TSwitch::fNetBandwidth. Only decrement the bytes if the head has been sent (indicating that the switch logic has executed).
[TPacketHead]
[TPacketEopGood]
[timeout]
[TPacketTail]
[TPacketAckNow]
[TPacketOkay]
Receive TPacketHead1 on TSwitch1.fInLink[n]
Receive TPacketEopGood1.
Receive TPacketTail1.
Receive TPackAckNow1.
Receive TPacketOkay1.
Create a TPacketReservation instance and queue it on the input queue for link n, virtual channel c.
Write TPacketGood1 to its out link.
Store TPacketTail1 in its TPacketReservation1 instance.
Store TPackAckNow1 in its TPacketReservation1 instance.
Write TPacketOkay1 to its inverse out link.
The fBytesRemaining attribute is set to the packet size.
Delete the TPacketReservation instance corresponding to TPacketGood1.
The virtual channel number is recorded in all outgoing packets.
Examine crossbar matrix to see if an out link is now available to send a new packet.
[out link available]
[out link unavailable]
Pop TPacketReservation1 off link input queue and put it in the crossbar matrix. Set TPacketReservation1.fDelay to TSwitch::fDelay to account for switching logic.
Examine all TPacketReservation instances to see if fDelay is now zero for any of them.
Examine all TPacketReservation instances to see if fAckBytes is now zero for any of them.
Examine all TPacketReservation instances to see if fBytesRemaining is now zero for any of them.
[fDelay == 0]
[fAckBytes == 0]
[fBytesRemaining ==0]
Remove TPacketHead1 from TPacketReservation1 and set its virtual channel.
Remove TPacketAckNow1 from TPacketReservation1 and set its virtual channel.
Remove TPacketTail1 from TPacketReservation1 and set its virtual channel.
Write TPacketHead1 to its out link.
Write TPacketAckNow1 to its out link.
Write TPacketTail1 to its out link.
Compute a new DeltaT based on the fDelay, fAckBytes, and fBytesRemaining attributes of the TPacketReservation instances in the crossbar matrix.
TNICReceiver2
Wait on TNIC2.fInNet.
[TPacketHead]
[TPacketAckNow]
[TPacketTail]
[TPacketGood]
Receive TPacketHead on TNIC2.fInNet.
Receive TPackAckNow on TNIC2.fInNet.
Receive TPacketTail on TNIC2.fInNet.
Receive TPacketGood on TNIC2.fInNet.
Create a TPacketReservation1 to hold the TPacketHead1 received.
Store the TPacketAckNow in TPacketReservation1.
Store the TPacketTail1 in TPacketReservation1.
Delete the TPacketReservation1 instance.
Write a new TPacketOkay on TNIC2.fOutNet.
Put the packet in TNIC2.fPacketBuffer.
Check to see if this is the last packet in the message.
[no]
[yes]
Reassemble TMessage1 from its packets.
Obtain a lock on TNIC2.fBus.
The DMA bandwidth may be asymmetrical.
Delay TMessage1.fSize / TNIC::fBusBandwidth seconds.
Release the TNIC2.fBus lock.
Write TMessage1 to TNIC2.fOutBus.

Figure 13: UML activity diagram outlining the processing of messages and packets in the medium-fidelity Quadrics network simulation.

2. The **TMessage** event moves across the link from **TSMPNode.fOutBus** to **TNIC.fInBus**, acquiring the delay specified in the DML. This represents the command code being written to the NIC's memory. The delay must equal the reciprocal of the PCI bus bandwidth.

3. The **TWorkloadListener** process queues the **TMessage** and signals the **TBusTransfer** process that a message is available. This represents the request for a DMA being queued.

4. The **TBusTransfer** process waits to obtain a lock on the **TNIC.fBusSem** semaphore. The semaphore accounts for the fact that the PCI bus is half-duplex. Obtaining the lock represents the first byte of the DMA transfer.

5. Once the **TBusTransfer** has the lock, it delays for **TMessage.fSize / TNIC.fBusBandwidth** seconds and then releases the lock. This write represents the last byte of the DMA transfer. The **TNIC.fBusBandwidth** parameter represents the bandwidth of the PCI bus. The **TBusTransfer** process pops the top **TMessage** off its queue of message requests and signals the **TNICSender** process that a message awaits.

6. The **TNICSender** packetizes the **TMessage** by creating a queue of **TPacketReservation** instances. When these **TPacketReservation** instances are created each holds a **TPacketHead**, **TPacketTail**, and **TPacketAckNow** event and has a **TPacketReservation.fBytesRemaining** attribute equal to the packet size. The **TPacketHead.fAckBytes** attribute should be set according to when $PACK\_OK$ should be sent by the receiver: this is the number of bytes that must be sent before **TPacketAckNow** is sent. We delay **TNIC.fMessageDelay** seconds here to account for the NIC logic that executes.

7. The **TNICSender** starts transmitting the packet at the top of its queue by writing the **TPacketHead** event to the **TNIC.fOutNet** channel after a delay of **TNIC.fPacketDelay** seconds. Before being written, **TPacket.fVirtualChannel** is set to zero. Since the NIC can only send on virtual channel 0, it does not have to share bandwidth between both virtual channels.

8. The **TNICSender** waits **TPacketReservation.fAckNowBytes / TNIC.fNetBandwidth** seconds and then writes the **TPacketAckNow** event to the **TNIC.fOutNet channel**. The **TNIC.fNetBandwidth** parameter represents the bandwidth of the network links.

9. The **TNICSender** waits **TPacketReservation.fBytesRemaining / TNIC.fNetBandwidth** seconds and then writes the **TPacketTail** event to the **TNIC.fOutNet channel**.

10. After **TPacketTail** has been sent and the **TPacketOkay** event from the **TNICReceiver** has been received, the **TNICSender** sends a **TPacketEopGood** event.

11. Meanwhile, the **TPacket...** events move from the **TNIC.fOutNet** channel to the **TSwitch.fInNet**, acquiring a delay specified in the DML. The delay corresponds to the physical speed of transmission in the cable.

12. When a **TPacketHead** event is received at a **TSwitch**, a **TPacketReservation** instance is created and queued on the input buffer corresponding to its input port and virtual channel. The **TPacketReservation** instance holds the **TPacketHead** and its **fBytesRemaining** field is set to **TPacketHead.fSize**. Each switch has 8×2 input buffers. This buffering represents the $STOPx$ and $STARTx$ of the flit-level protocol.

13. Whenever a **TSwitch** has queued a **TPacketHead** or handled a **TPacketEopGood** event (described below), it next checks if the **TPacketReservation** at the top of the corresponding input buffer can be served by seeing whether one of the virtual channels for its output port is free. If a

virtual channel is available for the output port, then a pointer to the **TPacketReservation** is set there and a delay of **TSwitch.fDelay** seconds is assigned to **TPacketReservation.fDelay** to account for switch logic. Note that a wildcard in the packet's route can be assigned to any upward output port/channel.

14. A **TSwitch** moves time forward with the help of **TCrossbarSwitchTimer** timers that are derived from the DaSSF EventTimer and have additional state information. Each **TPacketReservation** has one of these associated timers. The timer is initially set to expire after **TSwitch.fDelay** has elapsed for the reservation. The **TSwitch** waits for packets to arrive or for a timer to expire. The action taken when a timer expires depends on the current state of the packet reservation as detailed in the following steps.

15. If **TPacketReservation.fDelay** is zero, then remove the **TPacketHead** and write it to the proper **TSwitch.fOutLink** channel. Before being written, **TPacket.fVirtualChannel** is set to the correct number. Schedule its timer to expire when **TPacketReservation.fAckNowBytes** bytes have been written based upon the number of virtual channels currently in use. If two virtual channels are in use, reschedule the timer for the reservation on the other virtual channel based upon its remaining bytes and the now shared bandwidth. Go to step 14.

16. If **TPacketReservation.fAckNowBytes** is zero, then remove the **TPacketAckNow** if it exists and write it to the proper **TSwitch.fOutLink** channel. Schedule the timer to expire when **TPacketReservation.fBytesRemaining** bytes have been written. Go to step 14.

17. If **TPacketReservation.fBytesRemaining** is zero, then remove the **TPacketTail** if it exists and write it to the proper **TSwitch.fOutLink** channel. If the other virtual channel is still in use, reschedule the timer for the reservation using it to expire based upon its remaining bytes and the now non-shared bandwidth. Go to step 14.

18. If a **TPacketAckNow** event has been received, hold it in the corresponding **TPacketReservations** instance in the **TSwitch**. If the **TPacketReservation.fAckNowBytes** attribute is zero, then go to step 16; otherwise, go to step 14.

19. If a **TPacketTail** event has been received, hold it in the corresponding **TPacketReservations** instance in the **TSwitch**. If the **TPacketReservation.fBytesRemaining** attribute is zero, then go to step 17; otherwise, go to step 14.

20. If a **TPacketEopGood** event has been received, then delete the **TPacketReservation** instance and write the **TPacketEopGood** event to the proper **TSwitch.fOutLink channel**. Go to step 14.

21. If a **TPacketOkay** event has been received, then write it to the proper **TSwitch.fOutLink** channel of the inverse crossbar matrix. Go to step 14.

22. When the **TNICReceiver** process receives a **TPacketHead** event, it creates a **TPacketReservation** instance that holds the **TPacketHead**.

23. If a **TPacketAckNow** event has arrived, then hold this in the **TPacketReservation** instance and send a **TPacketOkay** event on the virtual channel **TPacket.fVirtualChannel**.

24. If a **TPacketTail** event has arrived, then hold this in the **TPacketReservation** instance. If this was the last packet in the message, go to step 26.

25. If a **TPacketEopGood** event has arrived, then delete the **TPacketReservation** instance.

26. Once the **TNICReceiver** has all of the packets, it recreates the **TMessage** and signals the **TBus-Transfer** process that a message is available.

27. The **TBusTransfer** process waits to obtain a lock on the **TNIC.fBusSem** semaphore. Once the **TBusTransfer** has the lock, it delays for **TMessage.fSize / TNIC.fBusBandwidth** seconds and then releases the lock. The **TMessage** is written to **TNIC.fOutBus**.

28. The **TWorkloadReceiver** process handles the receipt of the **TMessage**.

Our implementation of more advanced simulation output data collection features relies on a new **TDataCollector** class that mediates all types of data collection and error logging. In addition to the complete event detail available in the low-fidelity model—which tracks the history of messages and propagation of packets—it can support the collection of summary data which may be sliced into time windows. Summary data includes information on queue sizes, throughputs at switches, port usages, timeouts, and path lengths. The output is configurable so that it can be retrieved either as it is generated or after the simulation complete.

6.4 Testing

The medium fidelity network representation is considerably more complex than the low fidelity representation, with more options that require testing and involving a more tedious verification process. We determined that, as is often the case in software development projects, an automated regression testing capability would greatly facilitate the maintenance and enhancement of the medium fidelity network model in particular, and any number of more sophisticated models to be developed in the future.

The essence of the testing procedure is the maintenance of an archive of models, various parameterizations of each model, and for each of the latter the canonical or reference output against which output to be verified may be compared. The comparison function is arbitrary—programmatically defined, recognizing the fact that in the context of communicating processes with arbitrary synchronization the timing and ordering of messages, and the timing of many other events of interest, is not deterministic. Thus a significant task of the test designer is the identification of the intended invariant properties of a particular model (possibly as a function of its parameterization), and the codification of those invariants (effectively as equivalence classes) in the comparison function.

Full details of the regression testing are available elsewhere [24].

6.5 Scaling Behavior

In order to understand how our medium-fidelity network simulation scales as a function of the size of the machine on which it is run, we have performed direct execution simulations of the ASCI SWEEP3D application. Three partitions of a single SWEEP3D problem involving a $50 \times 50 \times 50$ cell computational grid: a 2×2 partition of computational nodes (with one node per CPU), a 3×4 partition, and a 6×6 partition. For each of these we varied the number of computational nodes (with one node per CPU again) used for the simulator and measured its performance. All simulations were run on the *wolverine* machine, *wms.acl.lanl.gov*.

Because there are large time delays between the initiation of a message send on a node and its actual departure from a NIC, it is most efficient to partition the simulation so that the network and NICs reside on the same DaSSF timeline (since they have short delays between each other). The nodes are evenly distributed among the remaining timelines. For all of the simulation runs we assign a single timeline to each computational node (CPU).

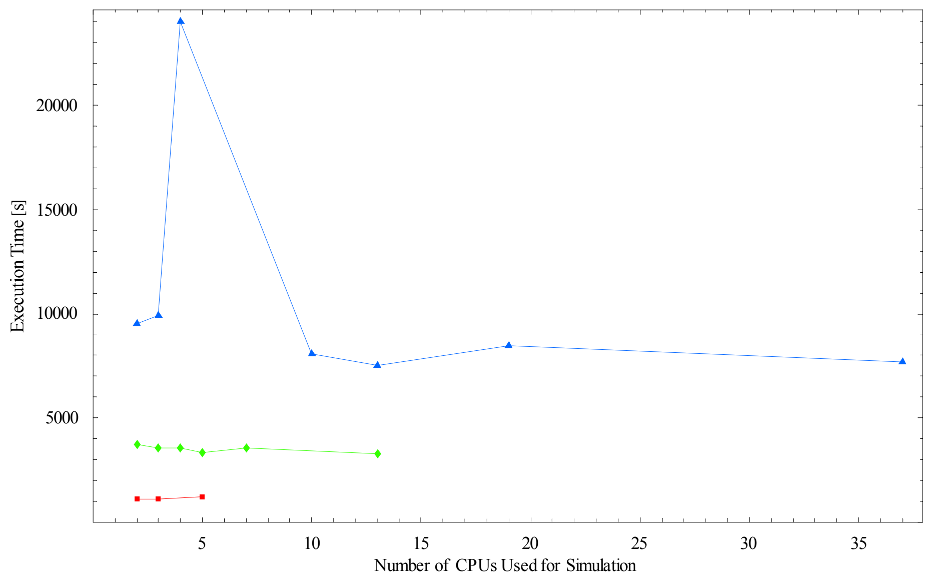

Figure 14: Execution time of the simulator (not the simulated application) as a function of the number of nodes (CPUs) on which it was run for three different partitionings of a SWEEP3D problem: 2 × 2 (red squares), 3 × 4 (green diamonds), and 6 × 6 (blue triangles).

Figures 14 through 17 illustrate the scaling behavior of the medium-fidelity simulation of this application. It is clear that the addition of computational nodes does not reduce the overall execution time (Fig. 14) or average CPU time (Fig. 15). We expect reductions in time to be apparent when much larger SWEEP3D problems are simulated, however. The peak memory usage (Fig. 16) of the simulation increases as a function of the number of simulation nodes because event queues tend to be larger when more timelines are present. Similarly, more kernel communication is necessary (Fig. 17) when more timelines are present.

6.6 Calibration & Validation

6.6.1 Network "Pings"

Before attempting to validate the medium-fidelity simulation against a real application, we need to determine the various parameters used by the simulation. Many of these can be set to values given in the network design specifications, but some must be found empirically. To do this, we consider the case of "pinging" one node with a message from another node. We examine three types of pings: (i) sending a message from the onboard memory of an Elan network interface card to another; (ii) sending a message from the main memory of a node to another; and (iii) sending a message using the MPI protocol, which may use additional buffers in main memory. For each of these cases we performed a large set of pings using different message sizes and passing through different portions of the interconnect network. All measurements were made on a 64-node Compaq machine (*wolverine*) with four ES-40 CPUs per node and a three-layer Quadrics network.

Table 6 shows the parameters that must be set in any simulation: the ones shown in boldface type were set by studying the results of regression analysis of ping measurements; the other parameters were determined from design documentation describing the Elan/Elite hardware. Figure 18 shows the raw data. We encountered several significant difficulties in determining these parameters:

1. There is an anomalous discontinuity in measured latencies for messages involving one versus

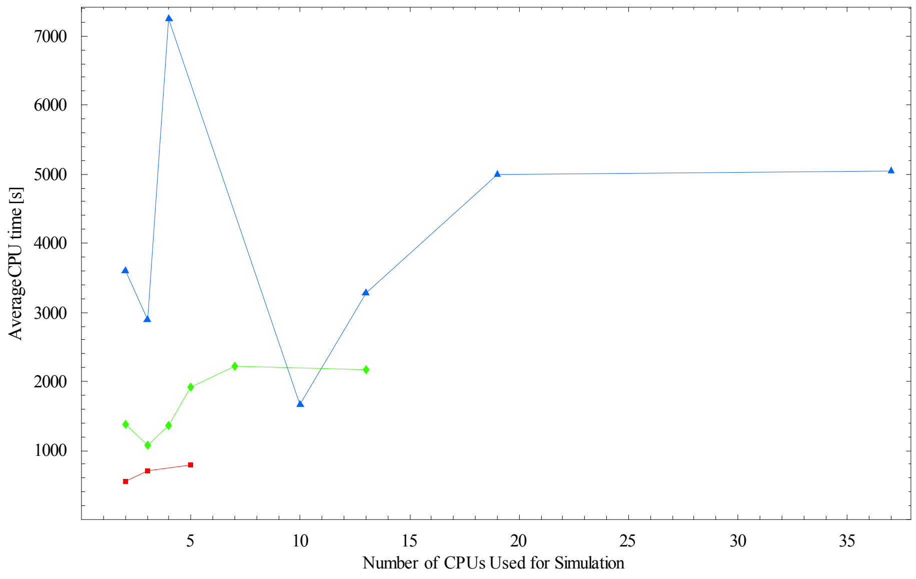

Figure 15: Average CPU time (i.e., total CPU time divided by the number of CPUs) used by the simulator (not the simulated application) as a function of the number of nodes (CPUs) on which it was run for three different partitionings of a SWEEP3D problem: 2 × 2 (red squares), 3 × 4 (green diamonds), and 6 × 6 (blue triangles).

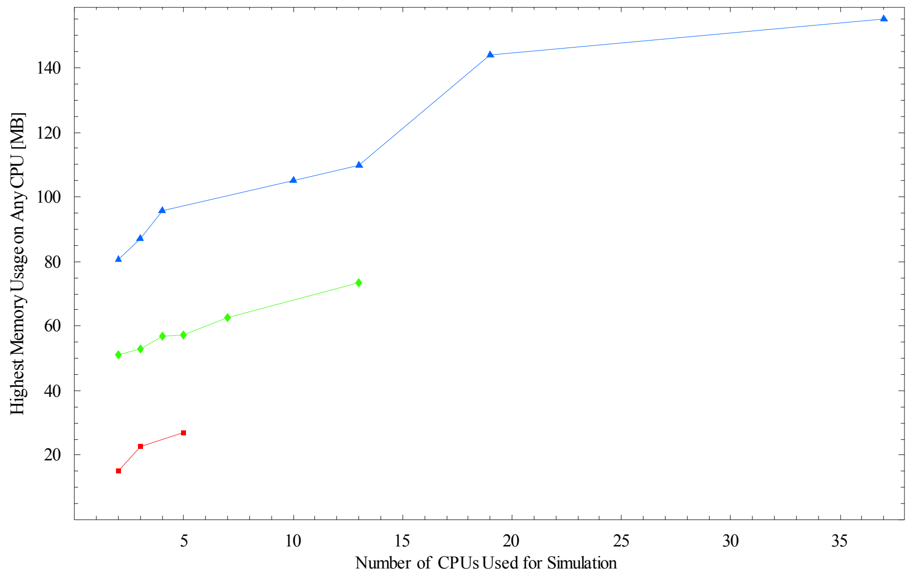

Figure 16: Peak memory usage on any CPU for the simulator (not the simulated application) as a function of the number of nodes (CPUs) on which it was run for three different partitionings of a SWEEP3D problem: 2 × 2 (red squares), 3 × 4 (green diamonds), and 6 × 6 (blue triangles).

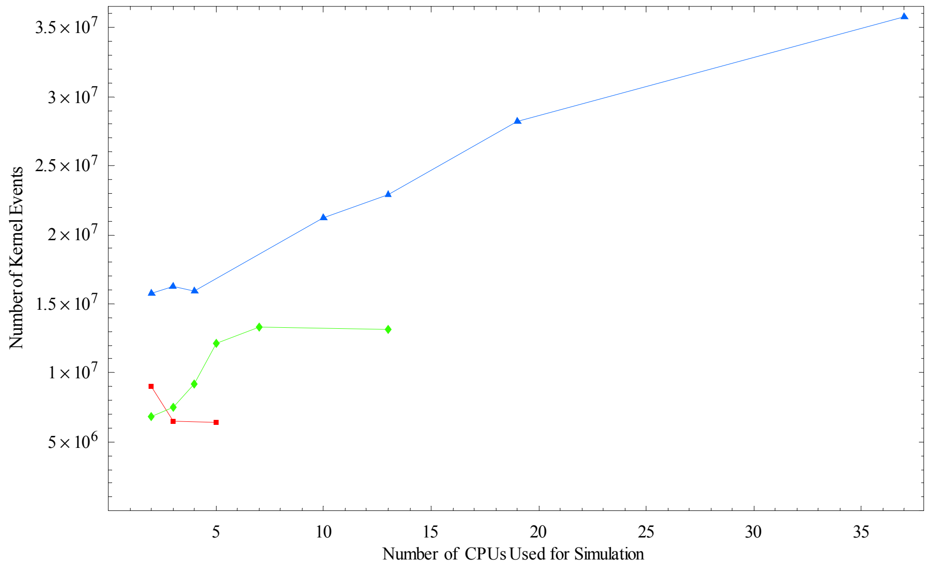

Figure 17: Number of simulation kernel events in the simulator (not the simulated application) as a function of the number of nodes (CPUs) on which it was run for three different partitionings of a SWEEP3D problem: 2 × 2 (red squares), 3 × 4 (green diamonds), and 6 × 6 (blue triangles).

location	parameter	Elan-Elan	Main-Main	MPI
NIC	MESSAGE_DELAY	**3.000 μs**	**3.750 μs**	**11.000 μs**
NIC	PACKET_DELAY	**120 ns**		
NIC	PACKET_SIZE	320 b		
NIC	ACK_BYTES	at end		
NIC	BUS_BANDWIDTH	∞	**800 MB/s**	**950 MB/s**
NIC, Switch	NET_BANDWIDTH	400 MB/s		
Switch	PACKET_DELAY	36 ns		
Node to NIC	channel delay	1 / BUS_BANDWIDTH		
NIC to Level 1 Switch	channel delay	**3 ns**		
Level 1 Switch to Level 2 Switch	channel delay	1 ns		
Level 2 Switch to Level 3 Switch	channel delay	**10 ns**		

Table 6: Parameters used in ping simulations: the parameters in boldface type were set from measurements, rather than from design specifications.

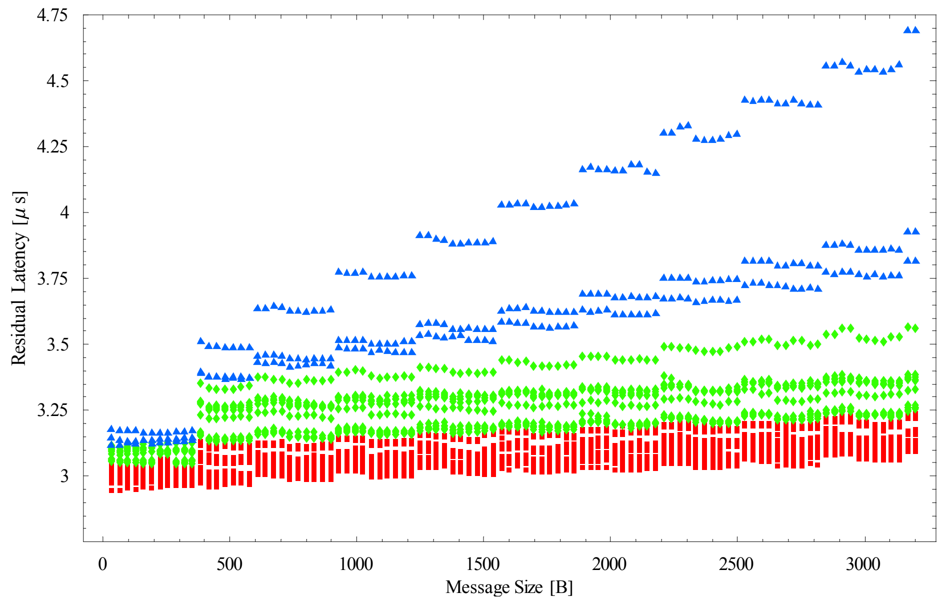

Figure 18: Residual latency (i.e., the measured latency minus the latency associated with the passage of the message and routing data over the network) measured on the *wolverine* machine: The red boxes represent messages passing only as far as the first level of the fat tree network; the green diamonds represent messages passing as far as the second level; and the blue triangles represent messages passing as far as the third level. Each data point represents the average of 200,000 ping measurements.

two packets if the packets have to pass above the first switch layer in the network. The Elan/Elite documentation [34, 35] gives no indication why such an effect exists. Since our simulation model was designed according to the Elan/Elite specification, we have had to modify the simulation by letting the PACKET_DELAY parameter at the NIC only take effect for the second packet of a message and only if that packet will pass through more than one switch.

2. The slopes of all three series of data are inconsistent with the implication in the Elan/Elite documentation that subsequent packets in a message incur the same switch delay as the first packet. The data are consistent with the assumption that subsequent packets incur no switching delay at all. We have modified our simulation to only use the PACKET_DELAY parameter for the first packet of a message passing through a switch.

3. The fact that the blue triangles in Fig. 18 form series with two different slopes suggests that there may be a hardware problem introducing additional latency for part in part of the third level of switches. In our analysis we have ignored the series with the higher latencies.

4. It appears that packets are delayed much more between switch layers 2 and 3 than between 1 and 2, even though only a short amount of wire connects adjacent layers—they all reside in the same chassis. Layers 2 and 3 are connected through a backplane, so it is possible that "throttling" may occur.

5. Although the nodes have a 200 MHz bus, data is transferred from the node to the NIC at an effectively higher rate, probably due to DMA transfers and network operations occurring

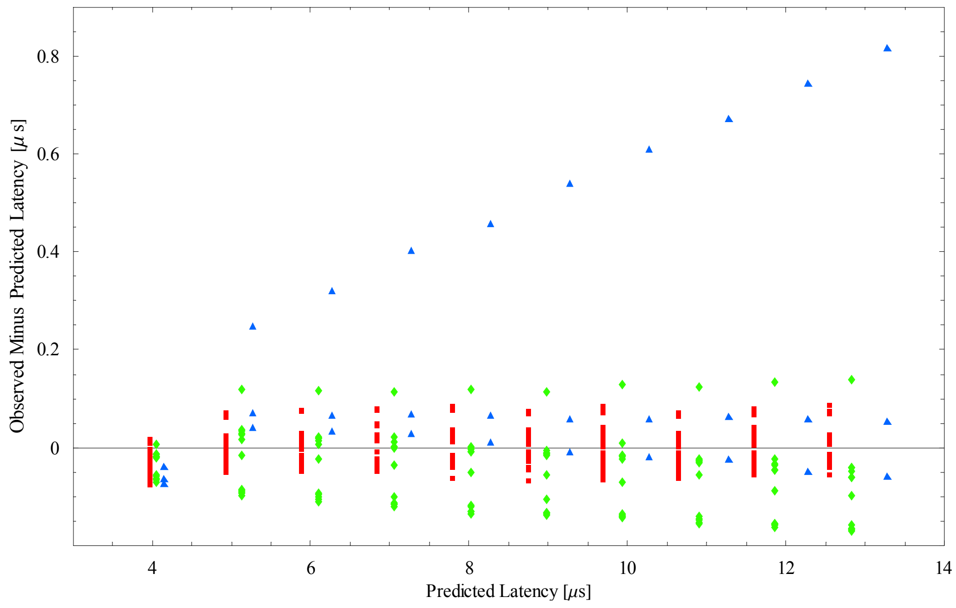

Figure 19: Error in predicting the latency of pings from Elan memory to Elan memory through a network of Elite switches: The red boxes represent messages passing only as far as the first level of the fat tree network; the green diamonds represent messages passing as far as the second level; and the blue triangles represent messages passing as far as the third level. Each data point represents the average of 200,000 ping measurements.

simultaneously. Since we do not have a detailed model for the PCI bus on these machines, we have opted to use the effective transfer rate in our model instead of the physical one.

It is possible points numbered 1, 2, and 4 above could have been addressed jointly with a single alteration in our simulation model, but we have not been successful in synthesizing one; it might be possible to model the data if the *ACK_NOW* token were acted on immediately with the receipt of the packet header, so that there would be no gap between the tail of the packet and the head of the next packet.

Figures 19–21 show the quality of agreement of our medium-fidelity ping simulations with measured data. For Elan-memory pings, we achieve about 100 ns accuracy; for main-memory and MPI pings, we achieve about 750 ns accuracy. The main-memory and MPI ping accuracy probably can only be improved by incorporating higher resolution memory and bus models into our simulation.

6.6.2 ASCI SWEEP3D Validation

In order to validate our simulation, we have compared the actual running time of the SWEEP3D application with a medium-fidelity simulation of it using the direct-execution workload. We consider the same three SWEEP3D partitions discussed in section 6.5. SWEEP3D and all of the simulations were run on the *wolverine* machine, *wms.acl.lanl.gov*; some of the simulations were run when this machine was loaded with other applications, so our measurements contain noise and systematic errors. A kernel patch supporting high-resolution timers was unavailable on this machine, so we had to rely on the standard Linux timing functions, which have a resolution of 1 ms. (We rounded times measured as zero up to 50 μs, based on our analysis in section 3.3.3.) We plan to rerun these experiments when a high-resolution timer is available and when the machine is unloaded.

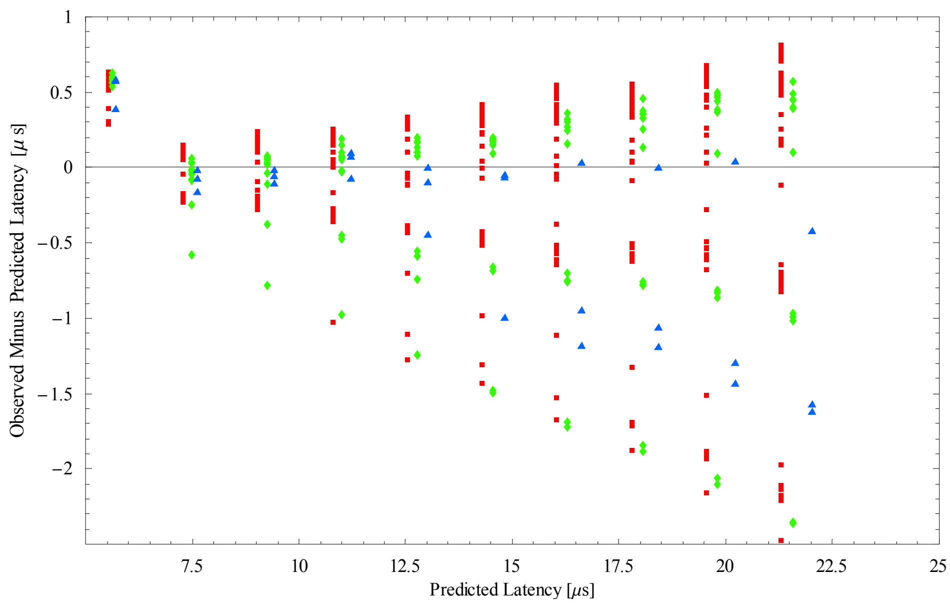

Figure 20: Error in predicting the latency of pings from main memory to main memory through a network of Elite switches: The red boxes represent messages passing only as far as the first level of the fat tree network; the green diamonds represent messages passing as far as the second level; and the blue triangles represent messages passing as far as the third level. Each data point represents the average of 200,000 ping measurements.

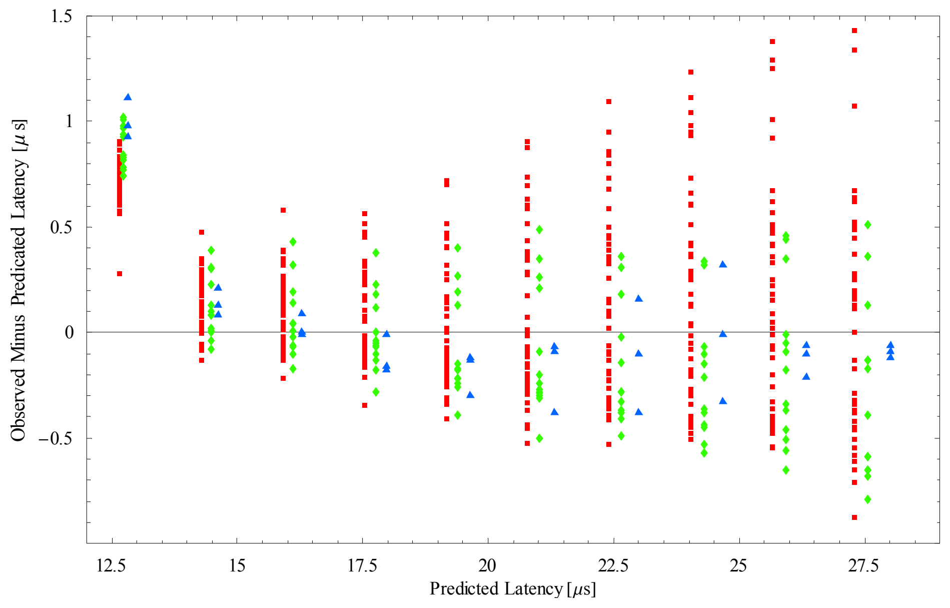

Figure 21: Error in predicting the latency of pings made with MPI through a network of Elite switches: The red boxes represent messages passing only as far as the first level of the fat tree network; the green diamonds represent messages passing as far as the second level; and the blue triangles represent messages passing as far as the third level. Each data point represents the average of 200,000 ping measurements.

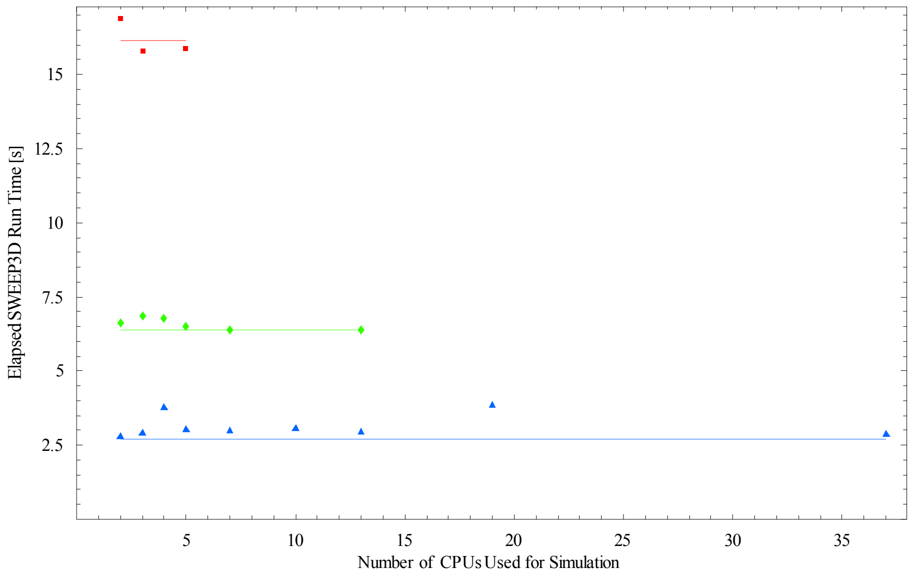

Figure 22: Comparison of SWEEP3D execution time with our simulation of it for three different partitions of a SWEEP3D problem: 2 × 2 (red, squares), 3 × 4 (green, diamonds), and 6 × 6 (blue, triangles). The points show the simulation's estimate of the SWEEP3D execution time and the lines show the actual execution time of SWEEP3D for the problem.

Figure 22 shows that our simulated execution time generally agrees with the actual execution time to within 10% (see Fig. 23). The couple of points with larger errors may have resulted from interference with other jobs running on the machine during the simulation. Note that no parameters in our network model were adjusted for this validation—all of the parameters were determined from the MPI ping analysis above.

7 Visualizing Simulated Fat-Tree Interconnect Networks

Our visualization efforts focus both on viewing the execution of the simulation and on displaying the performance of the simulated system. Visualization also aids in debugging the simulation itself, in developing and evaluating the efficiency of load balancing of the simulation entities, and in understanding synchronization between simulation timelines. Visualizing the simulated system allows end-users to understand how varying workload or network architecture affects the overall performance of an advanced or novel architecture. They can also see communication patterns, levels of network usages, and the presence of bottlenecks. Our visualization approaches include direct representations of the architecture as well as innovative abstractions of the architecture and dynamics of the system.

7.1 Flatland: An Immersive Visualization Development Framework

Flatland is an immersive visualization development framework created at the University of New Mexico as part of the Homunculus project [13]. It is used to facilitate rapid prototyping and research in scientific and information visualization, immersive environments and interfaces, and human factors engineering. The Flatland infrastructure aids in: creating and managing complex scene graphs with OpenGL geometry, lighting, shadows, stereo rendering, and spatialized sound

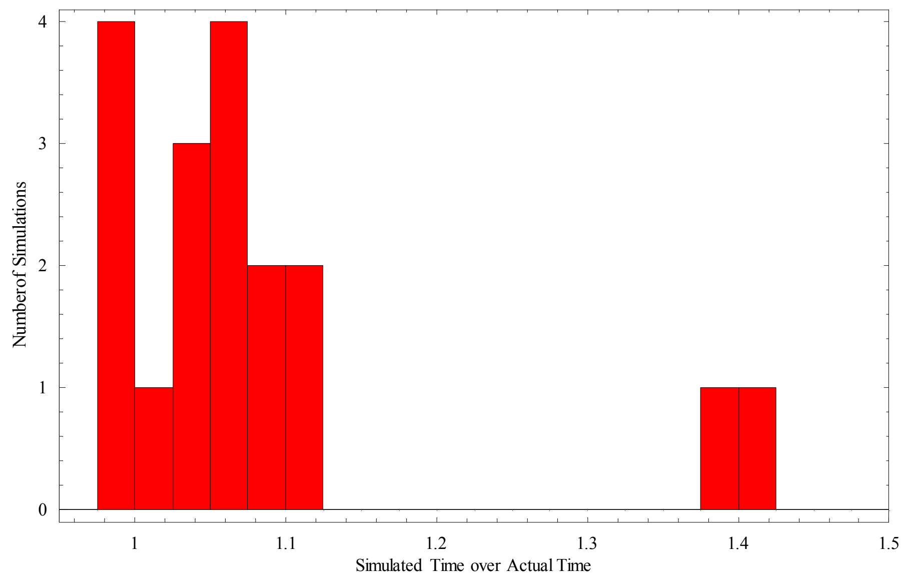

Figure 23: Comparison of SWEEP3D execution time with our simulation of it.

objects; dynamically loading applications in ways that allow them to interact without interfering; managing novel input and output devices; facilitating navigation of the resulting virtual spaces; and some basic, optional “world” objects such as the landscape, stars and sun that you see in Figure 24.

One of the tenets of our approach to visualization is that immersive, 3D environments can offer unique advantages over two dimensional plots, charts or graphs as well as simple, non-immersive three-dimensional graphics. By placing the user of these tools within the same context as the objects being viewed and allowing the user to navigate around them, picking their own points of view and using motion parallax to comprehend the three dimensional relationships between objects, allowing them to cast shadows and perhaps even to have behavior and emit sounds, a qualitative improvement in comprehension of the data is achieved.

7.2 Direct Representation

For purposes of debugging a simulation, it is useful to have a visual representation that clearly and explicitly represents all of the simulation's components. With these fat-tree-connected massively parallel computing architectures, we started by laying out the processors, NICs, and switches in several relatively obvious direct representations. Figure 25 shows a three-dimensional visualization of a 64-node network where the layers are arranged vertically so that the computational nodes are on the bottom and the successive layers of switches are arranged above one another. The first (Representation DL) is a simple distribution of processors and NICs along a line with the layers of switches above them also along a line. The second (Representation DC) simply involves wrapping this line around into a circle, creating a sort of cone topped with a cylinder. The third (Representation DR) is a rectangular grid laydown of the processors, NICs and switches. The representations shown are for a 64 processor machine with 3 layers of 16 switches each. Only the links for messages actively being transmitted through the network are drawn (and color coded according to message properties). The visualization is animated so that the user can see the progression of messages sent through the network.

Although the direct representations provide the highest level of detail concerning message prop-

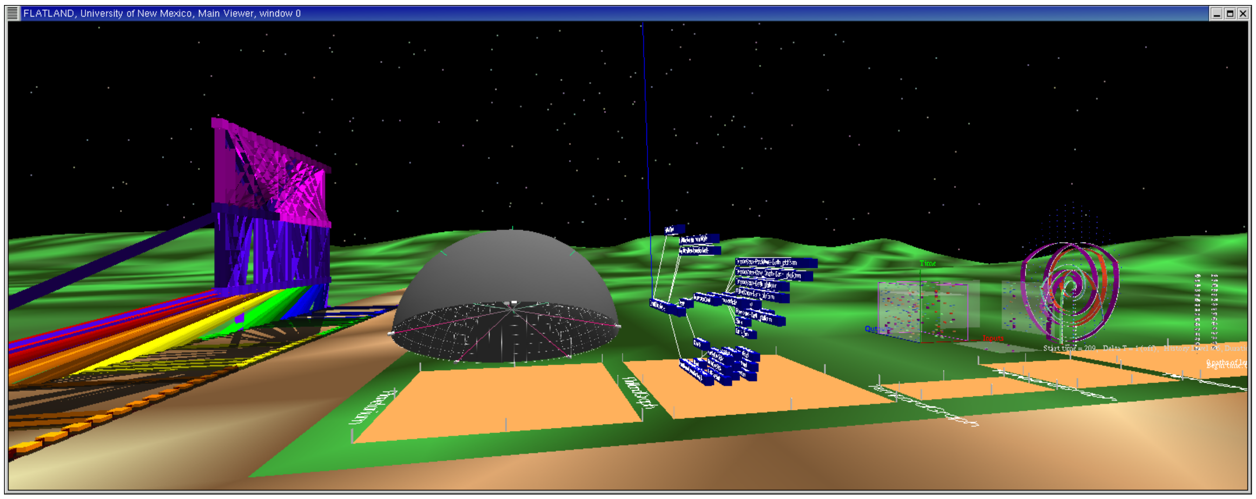

Figure 24: Flatland with a variety of independently developed applications from various problem domains.

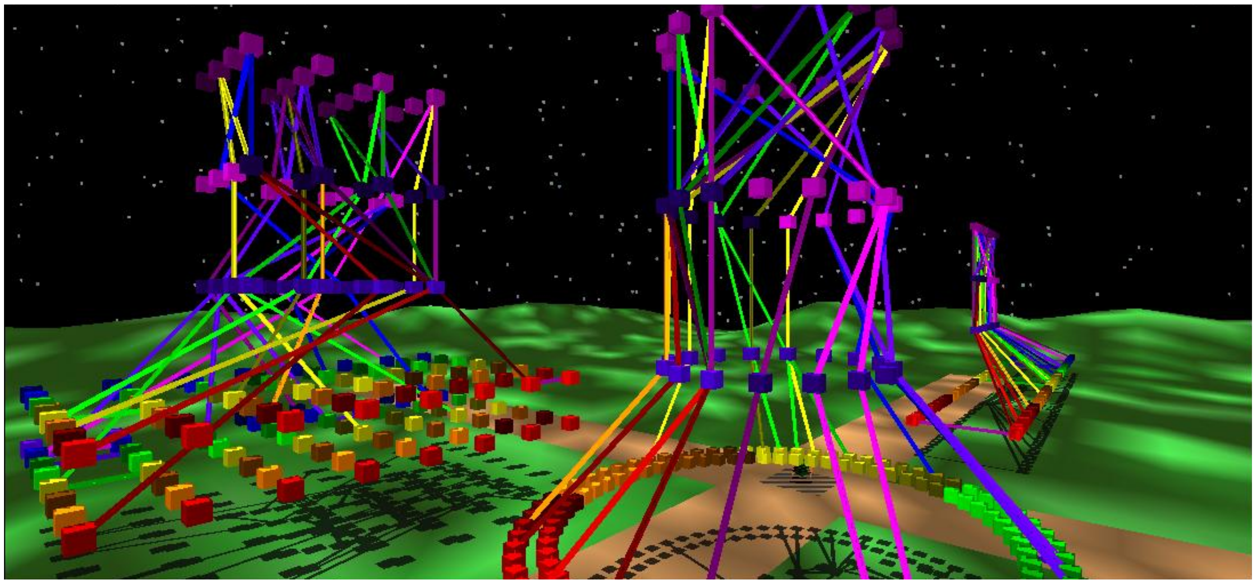

Figure 25: *Representation DR* (left), *Representation DC* (right, front), and *Representation DL* (right, rear): The bottom layer of boxes represents computational nodes and the upper layers of boxes correspond to the layers of switches. Only the links for messages currently in the network are drawn. The boxes and links are color-coded according to their attributes and the whole visualization is animated as simulation time progresses.

erties and component connectivity, they suffer from the lack of scalability and the visual complexity of the display (i.e., number of overlapping lines) when large models (more than several hundred computational nodes) are displayed. The *ad hoc* attribute coloring capabilities make this representation extremely valuable for debugging purposes, however. The largest machine we have studied is a 4096 processor machine with 6 layers of 1024 switches each. These representations were overwhelmed at that scale.

7.3 Layered Block Representation

In order to address the scalability and visual complexity issues of the direct representations, we have developed a more abstract representation (motivated by the abstraction of a graph as a connection, or adjacency, matrix) where switch layers are grouped together into aggregate visual objects. The whole network is laid out on a square with equally-spaced pillars along the diagonal representing the computational nodes and their NICs. The various switch layers are grouped by fours on succeeding levels below. For example, the first level below the nodes consists of groups of four switches, the second level consists of groups of sixteen switches, etc. Figure 26 shows a view of this representation. When a processor sends a message to another processor through the switching fabric, a line or "pipe" leaves the processor and grows across the switching fabric until it comes to the point on the implied connectivity matrix where the two processors' connection would normally be indicated. At that point it makes a 90 degree turn and continues to grow until it reaches the destination processor. In one mode of use, the layered switch blocks change color as they are involved in more and more communication traffic, allowing the viewer to recognize the relative level of utilization of each switch or switch group. It is also possible to subdivide a switch group vertically into individual switches.

The layered representation provides a more compact display than the direct representation and is somewhat more scalable, but it still suffers from similar problems once the number of nodes approaches one thousand. It does have the advantage of being able to show the activity in very large systems (e.g., 4096 computational nodes) if the pipes showing individual messages are suppressed. Unfortunately, the connectivity matrix scale is of order n_L, which naturally limits its scalability. At 4096 processors, individual processors, NICs, and even the first level of switches are smaller than a single pixel when viewed in their entirety, even on a high resolution screen (1600×1200). This is only a partial limitation since two dominant modes of use are likely to be macroscopic—attempting to understand the aggregate dynamics of the system which will be mostly differentiated at higher than NICs and first level of switches—or microscopic—focusing on following single messages through the system. Nevertheless, in this representation, full detail cannot be apprehended in full context simultaneously.

7.4 General Framework for Fat-Tree Representations

We now consider a general framework that allows us to express the layout of computational nodes and network switches in a two- or three-dimensional visualization of a fat-tree. It is actually sufficient to consider only switches in a layout, as each quadruplet of nodes connects to only a single switch—the lowest-level switch in any layout can be replaced by that switch and its four connected nodes. Likewise, we need not consider the eight duplex ports on each switch because each switch could be expanded into its eight *in* ports and/or its eight *out* ports in a layout.

Because some of these layouts have a fractal nature, we make a brief diversion to discuss the measurement of fractional dimension in these representations. Now we limit our consideration to two-dimensional layouts whose aspect ratio does not vary with L. Let w_L be the width of the

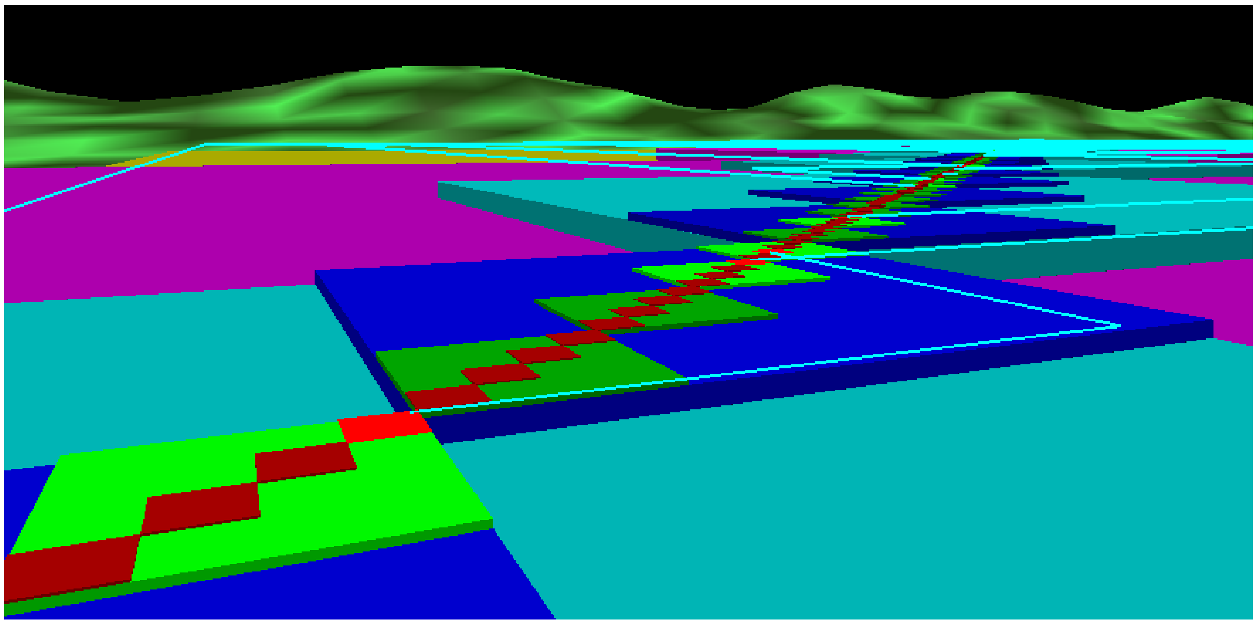

Figure 26: *Representation LB:* The network is laid out in a square. The first layer of switches is below this in groups of four (shown in red), the second layer of switches is below that in groups of sixteen (shown in green)—subsequent layers are blue (3rd), aqua (4th), magenta (5th), and yellow (6th). The switch groups brighten in color when one of their switches is active, and the "pipes" represent individual messages passing through the switches below them.

layout in pixels. The box-counting dimension d of the layout is defined from $s_L \sim {w_L}^d$ as $L \to \infty$. To compute this, we take a limit:

$$d = \lim_{L\to\infty} \frac{\log s_L}{\log w_L} = \lim_{L\to\infty} \frac{\log L + (L-1)\log 4}{\log w_L} = \lim_{L\to\infty} \frac{(L-1)\log 4}{\log w_L}. \tag{32}$$

Here is a general technique to represent the plethora of possible fat-tree layouts: Consider a pair of generating functions A_n and B_n which map the integers $0, 1, 2, 3$ to pixel coordinates; here n is a non-negative integer specifying the scale of the mapping. It is very important that the range of the functions A_n and B_n are disjoint—otherwise, switches on different layers will overlap on the same pixel. Since the fat tree is quaternary, it is useful to represent switch IDs in base four: namely, represent a number x as $x = \sum_{n=1}^{L-1} 4^{n-1} x_n$ where the x_n are its base-four digits. We can now write the pixel coordinates of switch x in layer ℓ out of a total of L layers as:

$$F_{L,\ell}(x) = \begin{cases} (0,0) & \text{for } L = 1 \\ \sum_{n=\ell}^{L-1} A_n(x_n) & \text{for } \ell = 1 \\ \sum_{n=\ell}^{L-1} A_n(x_n) + \sum_{n=1}^{\ell-1} B_n(x_n) & \text{for } 1 < \ell < L \\ \sum_{n=1}^{L-1} B_n(x_n) & \text{for } \ell = L \end{cases} \tag{33}$$

The layout is generated as the union of all the pixel coordinates of the switches:

$$\bigcup_{\ell=1}^{L} \bigcup_{x=0}^{4^{L-1}-1} F_{L,\ell}(x). \tag{34}$$

Note that although we have considered two-dimensional layouts here, the formalism extends trivially to the three-dimensional case.

7.5 Compact, Self-Similar, and Fractal Representations

At this point it should be clear that representations which essentially scale linearly with n_L are going to fail in at least the most obvious way of not providing full detail in full context. While level of detail management would help make this *dynamic range* problem somewhat more graceful, it would not allow us to apprehend each individual node, NIC, and switch, all at the same time without improving the *compactness* of our layout. The *direct conelike* representation described in Section 7.2 scales by approximately n_L, which is still linear with n_L but the *direct rectangular* layout scales by the $\sqrt{n_L}$. Thus 4096 processors, for example, only require an area on the order of 64×64 units to display. This seems very promising but by distributing the processors in a 64×64 array, the majority of the processors are in the middle of the area and similarly, the switching layers, laid out in 32×32 arrays tend to occlude each other nearly as badly. From this simple analysis, it appears that laying the processors, NICs, and each switch layer out in two dimensions is compact enough for our needs but leads immediately to occlusion problems.

Motivated by the somewhat self-similar nature of the fat tree, we began to investigate two different representations, one inspired by a simple pair of fractal generators as described in Section 7.5.1 and another inspired by *H-array radar antennae* as described in Section 7.5.2 below. The similarities between these two representations lead us to consider a more general representation of all layouts in two and three dimensions of fat trees.

7.5.1 Fractal Representation

We start with a fractal-based representation defined by

$$A_n(k) = 3^{n-1}a(k) \text{ and } B_n(k) = 3^{n-1}b(k) \tag{35}$$

where

$$a(k) = \begin{cases} (0,1) & \text{for } k = 0 \\ (1,0) & \text{for } k = 1 \\ (0,-1) & \text{for } k = 2 \\ (-1,0) & \text{for } k = 3 \end{cases} \text{ and } b(k) = \begin{cases} (-1,1) & \text{for } k = 0 \\ (1,1) & \text{for } k = 1 \\ (1,-1) & \text{for } k = 2 \\ (-1,-1) & \text{for } k = 3 \end{cases}. \tag{36}$$

The function $a(k)$ places the lower-layer switches on the sides of a square, while the function $b(k)$ places the higher-layer ones on the corners of the same square. The 3^{n-1} coefficient in Eq. (35) ensures that subsequent squares are appropriately scaled to a larger size. Figure 27 shows the fractal for the first several L—one can plainly see the self-similarity between networks of different sizes.

It is easy to see that $w_1 = 1$ and that $w_L = 3w_{L-1}$ for this representation. The solution to this recursion relation is $w_L = 3^{L-1}$, which results in a fractal dimension $d = \log 4/\log 3 \approx 1.26$.

This layout has the advantage that it scales well—one can represent the 6144 switches of a six-layer fat-tree in a 243×243 pixel area, for instance. It has a disadvantage that the switches for different layers are interleaved, making it somewhat tricky to visually separate the activity in different network layers. We have used animated versions of these layouts to successfully distinguish the distribution of messages in two 4096-computational-node applications with different communication patterns, however.

We can also examine the patterns of connections between switches at consecutive layers in this layout. Figure 28 illustrates this self-similar linking of switches.

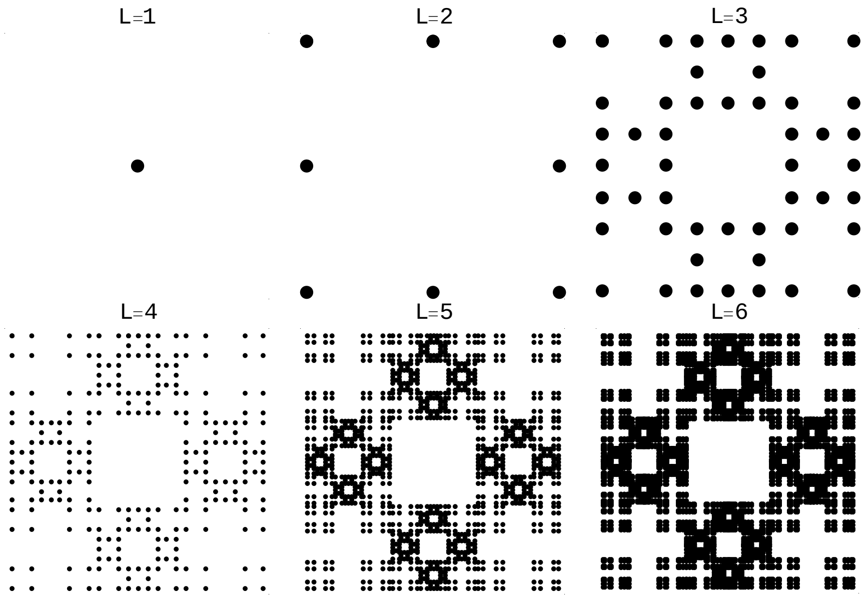

Figure 27: *Representation F1:* The six panels show placement of switches (circles) for the representation generated by Eq. (35) for $L = 1, \ldots, 6$.

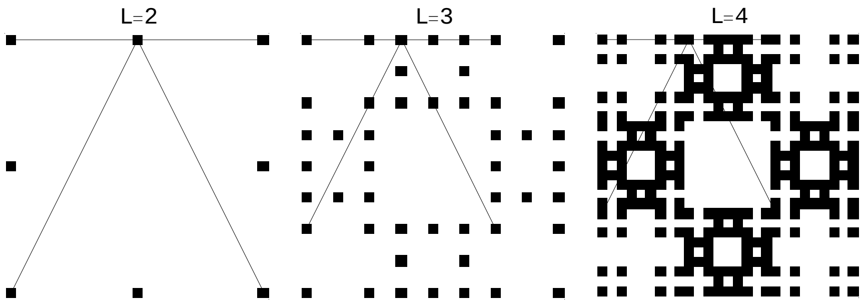

Figure 28: Connection patterns between consecutive layers of switches in Representation F1 (Fig. 27) for $L = 2, \ldots, 4$.

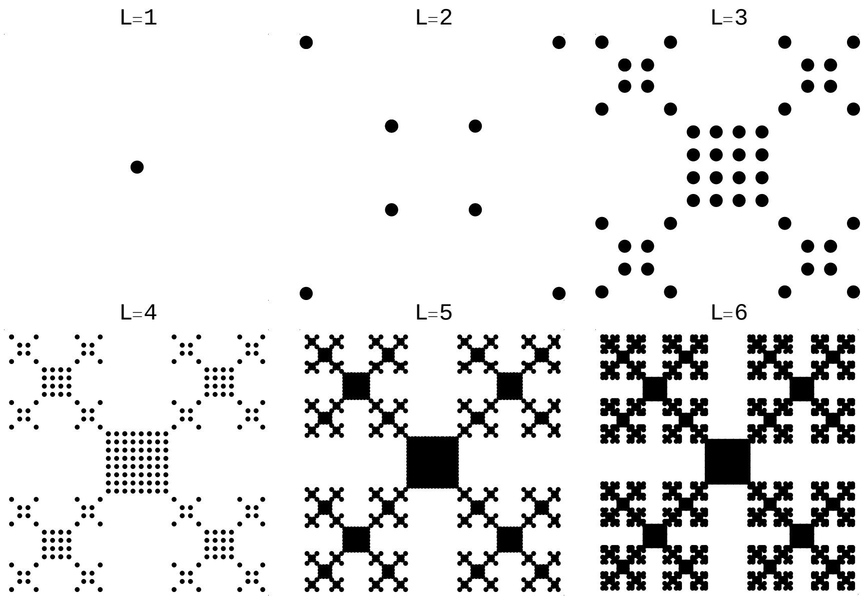

Figure 29: *Representation HT:* The six panels show placement of switches (circles) for the representation generated by Eq. (37) for $L = 1, \ldots, 6$.

7.5.2 "Fat H" Representation

We can address the problem of switch layers being interleaved in Representation F1 in a new representation defined by

$$A_n(k) = (n+2)2^{n-2}b(k) \text{ and } B_n(k) = 2^{n-2}b(k) \tag{37}$$

where

$$b(k) = \begin{cases} (-1,1) & \text{for } k=0 \\ (1,1) & \text{for } k=1 \\ (1,-1) & \text{for } k=2 \\ (-1,-1) & \text{for } k=3 \end{cases} \tag{38}$$

as in the other representation. In this case both the lower and upper level switches lie on the corners of a square, but the coefficient $(n+2)$ in Eq. (37) forces the lower level switches onto the corners of a larger square. Thus the more central groups of switches are in higher layers. Figure 29 illustrates this.

It is easy to see that $w_L = 1$ and that $w_L = 2w_{L-1} + 2^{L-1}$ for this representation. The solution to this recursion relation is $w_L = L \cdot 2^{L-1}$. This results in a fractal dimension $d = 2$. Hence, this representation "efficiently" fills two-dimensional space and is not technically a fractal (since it does not have fractional dimension). One can see this in Figure 29 in that the highest, central layer of switches occupies a smaller region of the diagram relative to the layers below as L increases. Hence, the layout is not self-similar as a function of L. One could rectify this situation by altering the scaling factors in Eq. (37) at the expense of letting higher layers take relatively more area in the diagram.

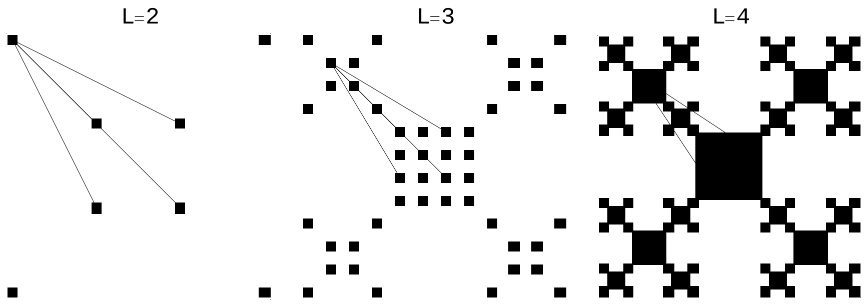

Figure 30: Connection patterns between consecutive layers of switches in Representation HT (Fig. 29) for $L = 2, \ldots, 4$.

The connection pattern between switches possesses a more straightforward arrangement in Representation HT (Fig. 30) than for the more interleaved Representation F1 (Fig. 28).

7.5.3 Additional Representations

We can also reformulate other representations in terms of the formalism of Equation (33). Representation QS (Fig. 5) can be expressed as

$$A_n(k) = \left(\left(k - \frac{3}{2}\right) 4^{n-1}, -\frac{1}{2}\right) \text{ and } B_n(k) = \left(\left(k - \frac{3}{2}\right) 4^{n-1}, \frac{1}{2}\right). \tag{39}$$

Representation LB (Fig. 26) can be written, more or less, as

$$A_n(k) = 4^{n-1}\left(k - \frac{3}{2}, k - \frac{3}{2}\right) \text{ and } B_n(k) = 4^{n-1}\left(k - \frac{3}{2}, (k + 2 \bmod 4) - \frac{3}{2}\right). \tag{40}$$

In general, to create additional representations, one only needs to choose a pair $(A_n(k), B_n(k))$ whose ranges do not overlap.

7.6 Fractal and H Tree Representation in Three Dimensions

We implemented the 2 dimensional self-similar representations F1 and HT in three dimensions. We then encoded the percentage of usage in height and color and added 3 dimensional connections between the switches. By representing connections with arcs whose height was proportional to the distance between two switches, communications over the connections were made unambiguous.

7.7 Floor Layout

In addition to the abstract layouts above, we have considered the physical layout of hardware in a machine room. These layouts are constrained by the size of the room, chassis configurations, and circuit board designs. Here we show an example layout for a machine with six switch layers, with 75% thinning between levels 5 and 6.

To set the scale for the layout, we define two parameters, $R = 30$ and $\rho = 6$. We define auxilliary variables r, θ, and ϕ:

$$r = \rho + x_2 + 4x_3 + I(x_3 \geq 2), \tag{41}$$

$$\theta = 2\pi \frac{x_4 + 4x_5}{16}, \tag{42}$$

$$\phi = 2\pi \frac{x_6}{3}. \tag{43}$$

where

$$I(z) = \begin{cases} 1 & \text{if } z \text{ is true} \\ 0 & \text{if } z \text{ is false} \end{cases}. \tag{44}$$

The three-dimensional location (X, Y, Z) of a node is:

$$X = R\cos\phi + r\cos\theta, \tag{45}$$

$$Y = R\sin\phi + r\sin\theta, \tag{46}$$

$$Z = x_1. \tag{47}$$

For switches in levels $\ell = 1, 2, 3$, we define:

$$r = \rho + 8 + \left(\frac{\ell}{2} - 1\right), \tag{48}$$

$$\theta = 2\pi \frac{x_3 + 4x_4}{16} + \frac{x_2 - \frac{3}{2}}{4r}, \tag{49}$$

$$\phi = 2\pi \frac{x_5}{3}. \tag{50}$$

The three-dimensional location (X, Y, Z) of such a switch is:

$$X = R\cos\phi + r\cos\theta, \tag{51}$$

$$Y = R\sin\phi + r\sin\theta, \tag{52}$$

$$Z = x_1. \tag{53}$$

For switches in levels $\ell = 4, 5$, we define:

$$\phi = 2\pi \frac{x_5}{3}. \tag{54}$$

The three-dimensional location (X, Y, Z) of such a switch is:

$$X = R\cos\phi + \left(x_3 - \frac{3}{2}\right) + \frac{x_2 - \frac{3}{2}}{4}, \tag{55}$$

$$Y = R\sin\phi + \frac{2I(\ell = 4) - 1}{4} + \left(x_4 - \frac{3}{2}\right), \tag{56}$$

$$Z = x_1. \tag{57}$$

For switches in level $\ell = 6$, the three-dimensional location (X, Y, Z) is:

$$X = \frac{2I(x_3 \geq 2) - 1}{2} + \frac{x_2 + 2(x_3 \bmod 2) - \frac{7}{2}}{8}, \tag{58}$$

$$Y = \left(x_4 - \frac{3}{2}\right), \tag{59}$$

$$Z = x_1. \tag{60}$$

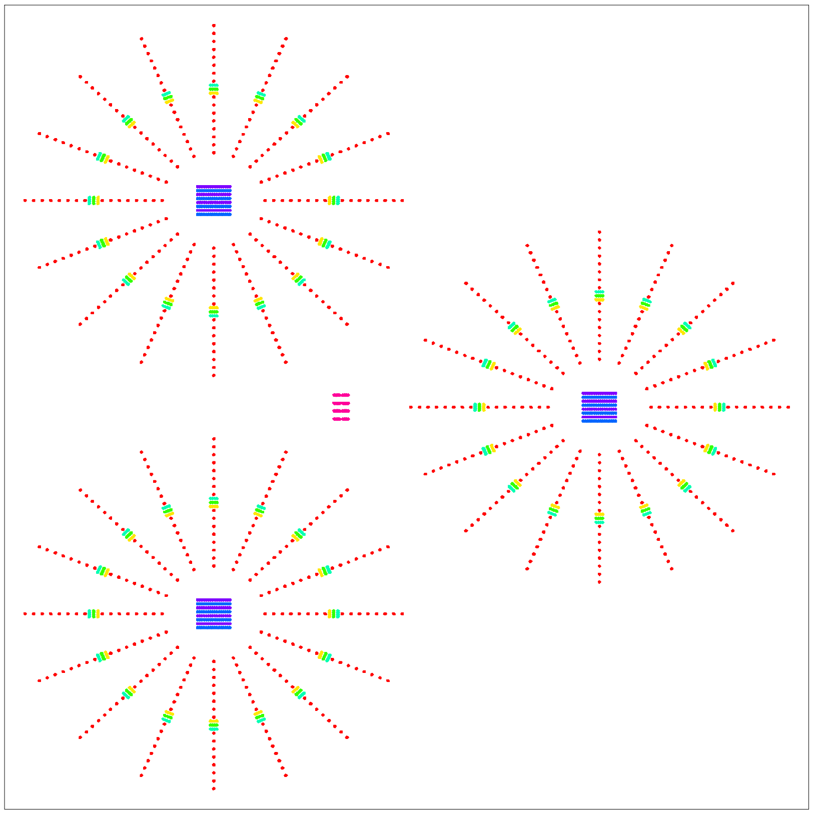

Figure 31: Example floor layout in two dimensions: six switch layers (differentiated by color) with 75% thinning between layers 5 and 6.

Figures 31 and 32 show this example layout in two and three dimensions. Because cable lengths vary from centimeters to decameters, this layout is impractical for visualization of traffic through the network cables. It can be used to give an idea of physical activity in the machine, however.

7.8 Comparison

It is instructive to compare the advantages and disadvantages of some of the fat-tree representations we have created so far. Table 7 contrasts the dimensions of Representations F1 and HT. Thus these have comparable sizes for the networks of interest ($L \leq 6$). Table 8 ranks all six representations in terms of usability metrics.

7.9 Visualization Prototype

The visualization tool under development for this project consists of a set of physical representations suitable for representing the topology of the network which in turn is used to register the events generated by the simulator or possibly a real system with proper instrumentation. By allowing an analyst to observe the abstract interactions and events within the simulation or the real machine, we expect to increase the understanding of the relationship between various events and states in the system.

Each representation we developed, starting with the suite of direct representations, then the

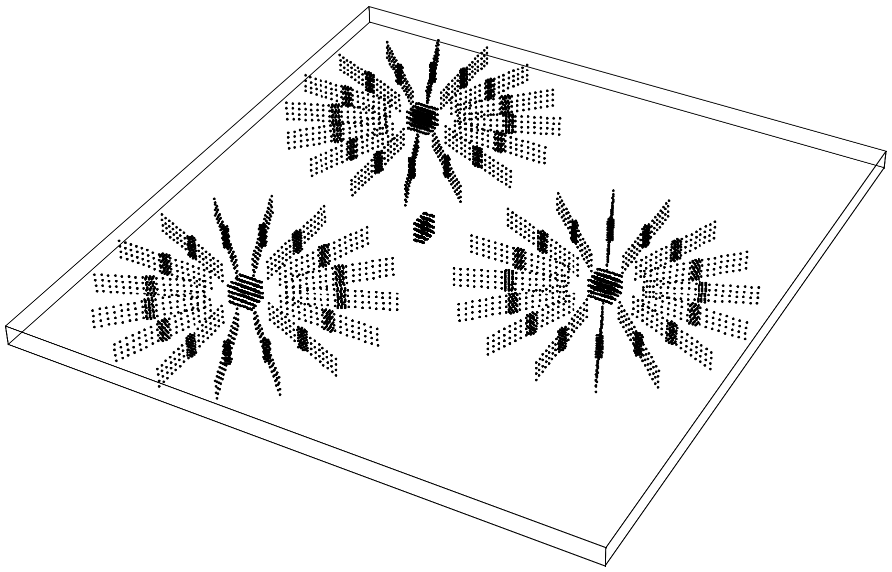

Figure 32: Example floor layout in three dimensions.

L	s_L	n_L	$w_L^{(\mathrm{F1})}$	$s_L / \left(w_L^{(\mathrm{F1})}\right)^2$	$w_L^{(\mathrm{HT})}$	$s_L \Big/ \left(w_L^{(\mathrm{HT})}\right)^2$
1	1	4	1	1.00	1	1.00
2	8	16	3	0.89	4	0.50
3	48	64	9	0.59	12	0.33
4	256	256	27	0.35	32	0.25
5	1280	1024	81	0.20	80	0.20
6	6144	4096	243	0.10	192	0.17

Table 7: Comparison of sizes of the layouts for Representations F1 and HT.

Property	QS	DL	DC	DR	LB	F1	HT
Microscopic Detail	very	very	very	very	yes	somewhat	somewhat
Scalability	no	no	somewhat	yes	somewhat	yes	very
Visual Complexity	very high	high	moderate	high	moderate	moderate	low

Table 8: Advantages and disadvantages of various representations.

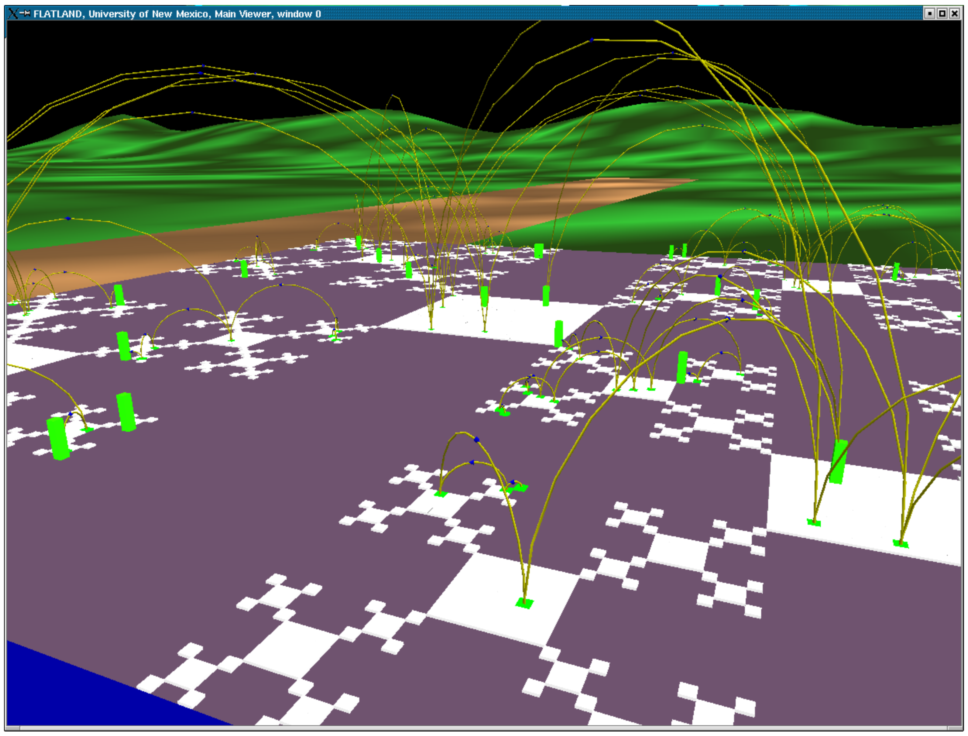

Figure 33: H Tree with connections represented as arcs

Layered Block (LB) and most recently the Fractal (F1) and self-similar H Tree (HT) was contrived to solve one or more obvious problems with the last. The direct representation was literally a first step, a place to start, where each node and switch was a cube and each communication between nodes and switches was a line or pipe. This lead to a great deal of visual clutter and while all three direct representations were laid out to minimize intersections between communications by arranging them in layers, there was no accommodation for the finer details of the topology. The Layered Block (LB) was an attempt to represent the hierarchical fat quad-tree topology in a variant of a connectivity or adjacency matrix. This layout succeeded to some extent, although it did not scale well and the connections, confined approximately to a plane intersected badly. The self-similar representations F1 and HT, took greater advantage of the topology of the network and used 2 dimensional space more efficiently.

All representations suffered from visual clutter and occlusion and intersection of connections. We experimented with different representations of the connections, as simple lines and cylinders hugging the surface of the switch structure, and as cylindrical arcs whose heights are proportional to their length. As can be seen in Figure 33, the use of arcs of different heights does help to show more intuitively where the flow of communication is occurring in the switching fabric, and particularly in the F1 and HT case, how expensive one communication is relative to another. Specifically, communication between within the same level yields tiny arcs while communication between any two levels, larger arcs, and the higher in the tree, the larger the arcs. Since the ultimate goal of communication between two switches is to facilitate communication between nodes at the lowest level of the switching fabric, the NICs themselves, communication at the higher levels in the fabric implies communication down through all levels, thus the larger arcs at the higher levels qualitatively match the larger amounts of latency and contention in the whole system.

We also experimented with nonlinear magnification techniques on the F1 and HT representations

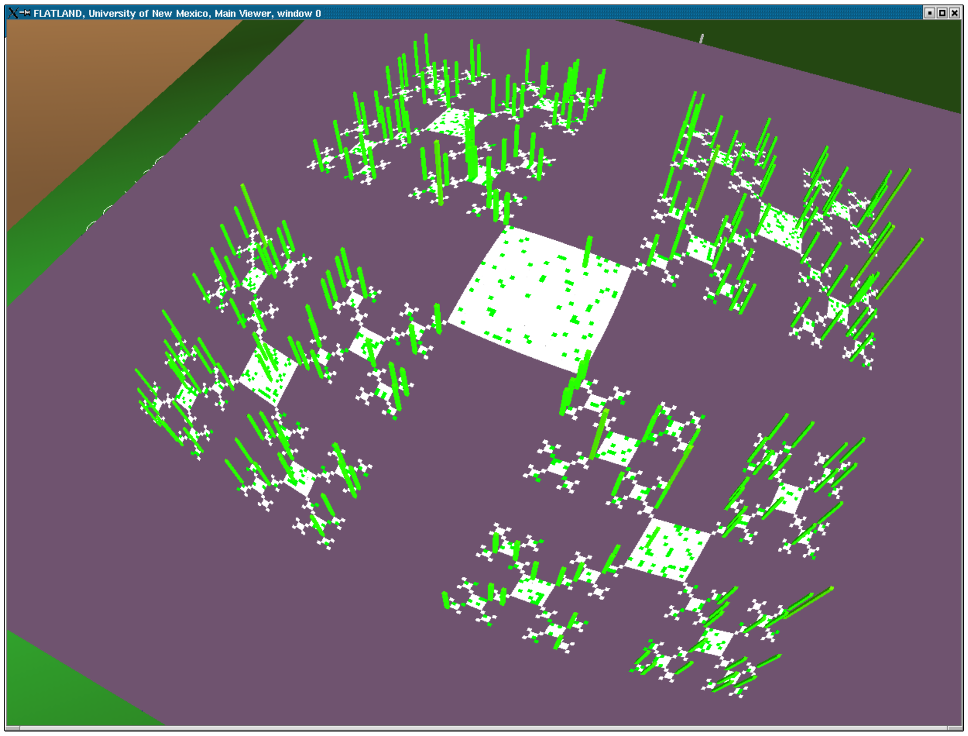

Figure 34: Representation HT with radial nonlinear magnification

(Fig. 34). In information visualization, the scale of the graphical objects affects the range of detail that is visible to the user. In this application, when the size of the network becomes large, the individual graphical elements representing switches and connections become difficult to see when the user attempts to observe the entire global representation. If the user zooms into a particular region of the graphic to get more detail, they lose the global context. Nonlinear magnification methods have been developed in the field of visualization to address this problem. In this method, the scale of the graphic is a nonlinear function of position in the field of view. Figure 34 illustrates this in the HT representation by increasing the magnification in the middle of the view while decreasing it around the edges. Thus a user can see the details at the focus while maintaining the global view at lower resolution.

The *à la carte* prototype visualizer currently provides the ability to read the data from a simulation run and run it forward and backward in time, controlling the speed of the animation of the events. It also allows the user to toggle on and off the various modes of the representations. In the Layered Block representation, this primarily means toggling on/off the visibility of the NICs, or toggling the connections between invisible, represented by a simple line following the surface, represented by a hollow pipe following the surface, and represented by a pipe in the shape of an arc, or toggling on and off the switch utilization as seen on the sides of the layers. In the Fractal and H Tree representations, along with the same time controls, the switch heights and colors can be toggled on and off and the connection modes can be toggled between invisible, straight lines on the surface, and arcs. A simple user's guide is under development which will outline the details of how to start the tool, select a given data set, control time and toggle the various representation modes. The basic setup of the visualizer is through the Flatland configuration file where the various applications that are to be run (or made available through a menu) are specified and the startup parameters are given. The interaction with a given application may be handled via a set of pull-

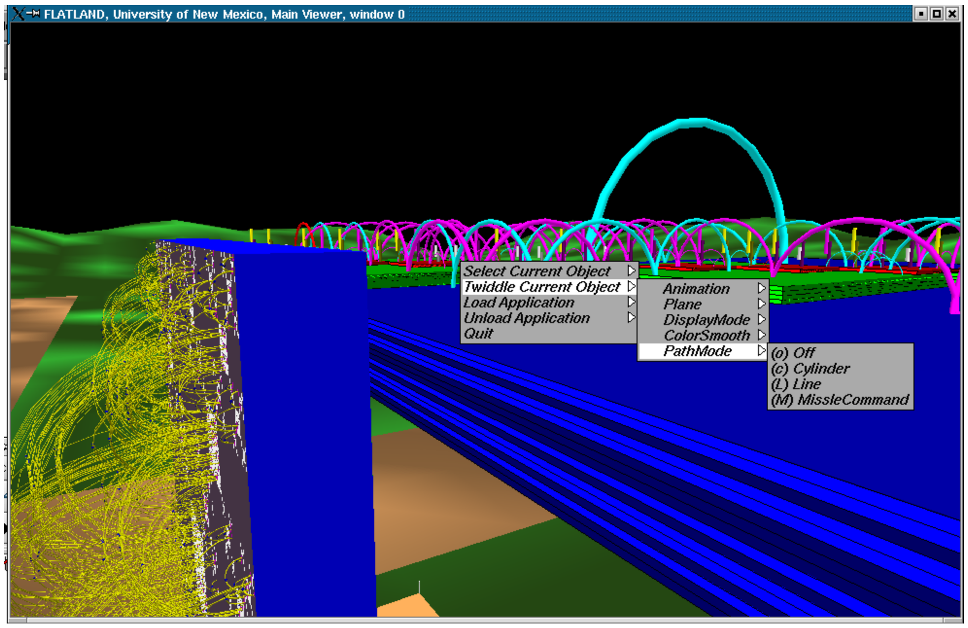

Figure 35: Representation HT with arc connections and height and color encoded switch usage.

down menus for each application, with the various options displayed in sub-menus, or through key bindings for each command in the menu. Figure 35 shows an example of the menus.

We have begun Phase II of the prototype visualizer which will include the ability to read machine topology specifications and lay out the nodes automatically and the ability to read Epilog format files for the event data. These capabilities will support the arbitrary machine topologies necessary for medium fidelity testing and validation. Secondly, they will allow us to use this visualizer on data collected from other sources that generate Epilog format data and will allow us to use other tools to analyze the output of the simulation. It is expected that the medium fidelity model will create new requirements for the visualizer and will suggest new representations and modes of interactions. To date, the visualizer has been only lightly used as the test cases have not been very interesting. The current activity of designing, building and debugging the medium fidelity model already has created a real demand for the prototype visualizer to aid in the debugging. The Phase II prototype is also being developed on top of a more general framework developed for another project. This framework separates the data reading and visual representation aspects cleanly and provides a separate module for mapping and controlling the representation of the data. This should shorten prototyping turnaround cycles, normalize the user interface, and reduce redundant development work when a new representation is desired.

8 Conclusion

We have outlined the design and implementation of our low- and medium-fidelity fat-tree network models and our low- and high-resolution workload representations along with the simulation and visualization technologies supporting them. Our component-based development process enables us to seamlessly compose hardware, protocols, and workloads of varying fidelities into a single simulation. These models have been calibrated to the behavior of an existing cluster computer of 64 nodes with 256 Alpha/Linux processors possessing an ELAN3 Quadrics network. The calibrated simulation accurately represents real MPI-based pings on the network to within about 750 ns; lower-level pings are modelled accurately to about 100 ns. We have validated our high-fidelity network and direct-execution workload modules against the real behavior of a representative ASCI application, SWEEP3D: we can predict the execution time of this application to within about 10%, even using the relatively course application timers available for our experiments. Our study of the scaling properties of our simulation indicates that it can handle much larger application instances than the SWEEP3D one used so far.

The major remaining risk in our effort is still that of scaling: Simulating extreme scale systems is resource-intensive even with efficient simulator fidelity. Constructing accurate medium-fidelity workload models is also a difficult process, and modeling operating system behavior might be required in some studies. Other risks areas are portability (accurately measuring directly- executed applications requires non-portable timers, and the modeling of low- level networking APIs may reduce portability) and validation (detailed measurements of applications on large machines are needed to validate a simulation).

9 Acknowledgements

This work was carried out under the auspices of the Department of Energy at Los Alamos National Laboratory under ASCI DisCom2. We would like to thank LANL's DisCom project leader, Stephen Turpin, for his support.

Thanks to members of the extended LANL ParSim team for technical input, including Mike Boorman, Fabrizio Petrini, and Harvey Wasserman; and to David Nicol and Jason Liu (Dartmouth College), and Josep Torrellas (University of Illinois at Urbana-Champaign). Anand Sivasubramaniam (Pennsylvania State University) and Madhav Marathe participated in the initial project proposal.

10 Appendices

10.1 DML Usage

This appendix describes the DML (Domain Modeling Language) keys and values we use to describe a simulated hardware architecture.

10.1.1 `ENVIRONMENT` Section

The key `LOG_STYLE` specifies how log messages are output. Valid values for this are `"OFF"`, `"IMMEDIATE"`, or `"AT WRAPUP"`. The default is:

```
LOG_STYLE    "IMMEDIATE"
```

The key `LOG_MASK` specified which log messages are output. Valid values for this are any subset of `"ERROR WARNING INFO DEBUG"` (it is space-delimited). The default is:

```
LOG_MASK     "ERROR WARNING INFO"
```

The keys `DATA_STYLE` and `DATA_MASK` act like those above, but for the data stream. They have defaults:

```
DATA_STYLE   "AT WRAPUP"
DATA_MASK    "INFO"
```

The key `EPILOG_STYLE` acts like the style keys above, but has valid values `"OFF"` or `"IMMEDIATE"`. It has the following default:

```
EPILOG_STYLE "IMMEDIATE"
```

Do not attempt to set the value to `"AT WRAPUP"`.

The key `EPILOG_FILE` specifies the file name for EPILOG output. It has no default.

The key `WORKLOAD` specifies the type of workload for the simulation. Valid values are `"StatWorkload"`, `"DirectWorkload"`, `"PingWorkload"`, or `"TestWorkload"`. Is has no default.

The key `NETWORK` specifies the type of network for the simulation. Valid values are `"LFQuadrics"` or `"MFQuadrics"`. It has no default.

The keys `FIRST_NODE`, `FIRST_NIC`, and `FIRST_SWITCH` provide the global IDs of the lowest-numbered node, NIC, or switch, respectively.

The key `MPI_COMM_SIZE` specifies the number of MPI nodes (i.e., the value returned by a call to MPI_COMM_SIZE with `MPI_COMM_WORLD` as its first argument).

The key `WOLVERINE` specifies whether to run the simulation in "wolverine compatibility mode" where delays are slightly different from the standard simulation. This key takes the values `"0"` and `"1"`.

10.1.2 `SMPNode` Section

The `PARAMS` section for each node has the following entries:

```
INT    # globally-unique integer identifying the entity
INT    # value returned by calls to MPI_COMM_RANK when
       # MPI_COMM_WORLD is its first argument
STRING # the network address of the node
```

The `CONFIGURE` section for each node has the following entries:

```
#### REQUIRED FOR StatWorkload SIMULATIONS ####
MESSAGE_SIZE  INT    # the mean message size [B]
MESSAGE_DELAY INT    # the mean delay between messages
TIME_LIMIT    INT    # the time [ns] after which messages
                     # are no longer sent
TARGET        STRING # the first possible target for
                     # messages
...                  # subsequent possible targets for
                     # messages
MAXTARGET     STRING # the maximum value of a possible
                     # target for messages when no
                     # TARGET keys exist

#### REQUIRED FOR DirectWorkload SIMULATIONS ####
EXECUTABLE    STRING # path to the executable program
TARGET        STRING # the first possible target for
                     # messages
...                  # subsequent possible targets for
                     # messages
MAXTARGET     STRING # the maximum value of a possible
```

```
                     # target for messages when no
                     # TARGET keys exist

#### REQUIRED FOR PingWorkload SIMULATIONS ####
MESSAGE_SIZE  INT    # the exact size [B] of the message
MESSAGE_DELAY INT    # the exact delay [B] between messages
PING_COUNT    INT    # the number of messages to send
TARGET        STRING # the first possible target for
                     # messages

#### REQUIRED FOR TestWorkload SIMULATIONS ####
MESSAGES      [      # the messages to be sent
  MESSAGE     [      # a message
    TIME      INT    # the time [ns] at which the message
                     # is to be sent
    TARGET    STRING # the target to which the message is
                     # to be sent
    SIZE      INT    # the size [B] of the message
    DATA      STRING # the data to be sent as the message
                     # (optional)
              ]      #
  ...                # additional messages
              ]      #
```

10.1.3 NIC Section

The PARAMS section for each NIC has the following entries:

```
INT    # globally-unique integer identifying the entity
STRING # the network address of the NIC's node
INT    # number of virtual channels
```

The CONFIGURE section for each NIC has the following entries:

```
#### REQUIRED FOR ALL SIMULATIONS ####
MESSAGE_DELAY INT    # delay [ns] between the receipt of a
                     # message and the start of packets
                     # being sent
PACKET_DELAY  INT    # delay [ns] between consecutive packets
                     # in a message
PACKET_SIZE   INT    # maximum packet size [B]
METHOD        STRING # the routing method

#### REQUIRED FOR MFQuadrics SIMULATIONS ####
BUS_BANDWIDTH INT    # the PCI bus bandwidth [MB/s]
NET_BANDWIDTH INT    # the network bandwidth [MB/s]
ACK_BYTES     INT    # the number of bytes sent before the
                     # ACK_NOW token
```

```
#### REQUIRED FOR LFQuadrics SIMLUATIONS ####
TIMEOUT       INT    # the timeout [ns] for retransmission

#### REQUIRED WHEN METHOD = "ReadRoute" ####
ROUTE         [      # the routing table
  TARGET      STRING #   the target address
  PATH        [      #   the path to the target
    PORT      INT    #     the port to use at the switch
    ...              #     subsequent ports
              ]      #
  ...                #   additional targets and paths
              ]      #
```

10.1.4 `Switch` Section

The `PARAMS` section for each switch has the following entries:

```
INT    # globally-unique integer identifying the entity
INT    # number of ports
INT    # number of virtual channels
```

The `CONFIGURE` section for each switch has the following entries:

```
#### REQUIRED FOR ALL SIMULATIONS ####
PACKET_DELAY  INT    # delay [ns] between consecutive packets
                     # in a message

#### REQUIRED FOR MFQuadrics SIMULATIONS ####
NET_BANDWIDTH INT    # the network bandwidth [MB/s]
```

10.2 EPILOG Usage

This appendix describes our usage of the EPILOG output format [40] used in our medium-fidelity simulation.

10.2.1 General

The global ID for an entity must be in the range [0, 999999].

10.2.2 SMPNode

All of the nodes are part of the `ELG_MACHINE` with identifier 0. The `ELG_NODE` identifier of the node is equal to its global ID in the DML. The `ELG_PROCESS` and `ELG_THREAD` identifiers in `ELG_LOCATION` are not used for nodes, and so are set to zero.

10.2.3 NIC

All of the NICs are part of the `ELG_MACHINE` with identifier 1. The `ELG_NODE` identifier of the NIC is equal to its global ID in the DML. The `ELG_PROCESS` identifier in `ELG_LOCATION` is not used, and so is set to zero. The `ELG_THREAD` identifier in `ELG_LOCATION` equals the channel number on the NIC.

10.2.4 Switch

All of the switches are part of the ELG_MACHINE with identifier 2. The ELG_NODE identifier of the switch is equal to its global ID in the DML. The ELG_PROCESS identifier in ELG_LOCATION equals the port number on the switch. The ELG_THREAD identifier in ELG_LOCATION equals the channel number on the switch.

10.2.5 Messages

The sending or receiving of a message by a Node is represented by a MPI_SEND or MPI_RECEIVE record, respectively. The location and destination/source identifier fields equal the numeric address of the source and target for the message. The communicator identifier field is always zero, and the message tag field is the numeric id of the message.

10.2.6 Packets

The sending or receiving of a packet by a NIC or switch is represented by a MPI_SEND or MPI_RECEIVE record, respectively. The destination/source identifier field equals the numeric address of the source of the message to which the packet belongs. The communication identifier field equals the numeric id of the message. The message tag field equals ten times the packet id plus type packet type (Head = 1, Tail = 2, AckNow = 3, Okay = 4, EopGood = 5). The message length is only reported when the head is sent. Note that these records need to be postprocessed before they can be used in a visualization tool like VAMPIR because we have misused the destination/source fields.

References

[1] F. J. Alexander and G. L. Eyink. A Rayleigh-Ritz Calculation of the Effective Potential far from Equilibrium. *Physical Review Letters*, 78(1), 1997.

[2] R. Bagrodia, E. Deelman, and T. Phan. Parallel simulation of Large-Scale Parallel Applications. *International Journal of High Performance Computing Applications*, 15(1):3–12, Spring 2001.

[3] R. Bagrodia, R. Meyer, M., Takai, Y. Chen, X. Zheng, J. Martin, and H. Song. Parsec: A Parallel simulation environment for complex systems. In *IEEE Computer*, 1998.

[4] R. L. Bagrodia. Parallel Languages for Discrete-Event Simulation Models. *IEEE Computational Science & Engineering*, Apr-June 1998.

[5] E. Benson. *3072-Node Quadrics Switch Topology.*

[6] E. Benson. *30T Overview.*

[7] L. Breslau, D. Estrin, K. Fall, S. Floyd, J. Heidemann, A. Helmy, P. Huang, S. McCanne, K. Varadhan, Y. Xu, and H. Yu. Advances in Network Simulation. *IEEE Computer*, 33(5):59–67, May 2000.

[8] Brian W. Bush. Idealized Q Layout. Los Alamos National Laboratory, 2002.

[9] James H. Cowie, David M. Nicol, and Andy T. Ogielski. Modeling the Global Internet. *Computing in Science & Engineering*, 1(1):30–38, 1999.

[10] D. E. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eiken. LogP: Towards a Realistic Model of Parallel Computation. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, San Diego, CA, May 1993.

[11] S. Das, R. Fujimoto, K. Panesar, D. Allison, and M. Hybinette. Gtw: A time warp system for shared memory multiprocessors. In *Winter Simulation Conference*, 1994.

[12] `http://www.cs.dartmouth.edu/research/DaSSF`.

[13] `http://www.ahpcc.unm.edu/homunculus/indexold.html`.

[14] Richard M. Fujimoto. *Parallel and Distributed Simulation Systems*. John Wiley & Sons, Inc., 2000.

[15] M. Goudreau, K. Lang, S. Rao, T. Suel, and T. Tsantilas. Towards Efficiency and Portability: Programming with the BSP Model. In *Proceedings of the 8th Annual ACM Symposium on Parallel Algorithms and Architectures*, Padova, Italy, 1996.

[16] `http://www.cc.gatech.edu/computing/pads/tech-parallel-gtw.html`.

[17] Adolfy Hoisie, Olaf Lubeck, and Harvey Wasserman. Performance Analysis of Wavefront Algorithms on Very-Large Scale Distributed Systems. In *Lecture Notes in Control and Information Sciences 249*. Springer Verlag, 1999.

[18] Adolfy Hoisie, Olaf Lubeck, and Harvey Wasserman. Scalability Analysis of Multidimensional Wavefront Algorithms on Large-Scale SMP Clusters. In *Proceedings of the 7th Symposium on the Frontiers of Massively Parallel Computations, Frontiers*, February 1999.

[19] R. Kaufman. The Q Supercomputer and Compaq. `http://www.compaq.com/hpc/news/news\_hpc\_60171.html`, November 2000. High Performance Technical Computing News, Issue 18.

[20] J. Liu, D. Nicol, B. Premore, and A. Poplawski. Performance Prediction of a Parallel Simulator. In *Proceedings of the Parallel and Distributed Simulation Conference*, Atlanta, GA, 1999.

[21] Jason Liu and David M. Nicol. *Dartmouth Scalable Simulation Framework User's Manual*. Dartmouth College Dept. of Computer Science, Feb 6 2002.

[22] Y.-H. Low, C.-C. Lim, W. Cai, S.-Y. Huang, W.-J. Hsu, S. Jain, and S. Turner. Survey of Languages and Runtime Libraries for Parallel Discrete-Event Simulation. *Simulation*, 73(3):170–186, March 1999.

[23] Nick Moss. A Compact Machine Representation Language for Simulation of Large-Scale Parallel Architectures. Technical Report LA-UR 02-6058, Los Alamos National Laboratory, 2002.

[24] Nick Moss. An Automated Regression Testing and Verification System for ParSim (Parallel Architecture Simulation Framework). Technical Report LA-UR 02-6057, Los Alamos National Laboratory, 2002.

[25] S. Mukherjee, S. Reinhardt, B. Falsafi, M. Litzkow, M. Hill, D. Wood, S. Huss-Lederman, and J. Larus. Wisconsin Wind Tunnel II: A Fast, Portable Parallel Architecture Simulator. *IEEE Concurrency*, Oct-Dec 2000.

[26] David M. Nicol and Jason Liu. Composite Synchronization in Parallel Discrete-Event Simulation. *IEEE Transactions on Parallel and Distributed Systems*, 13(5), May 2002.

[27] `http://www.isi.edu/nsnam/ns`.

[28] `http://icl.cs.utk.edu/projects/papi`.

[29] `http://www.fz-juelich.de/zam/PCL`.

[30] `http://www.cc.gatech.edu/computing/compass/pdns`.

[31] S. Prakash and R. Bagrodia. MPI-SIM: Using parallel simulation to evaluate MPI programs. In *Winter Simulation Conference*, 1998.

[32] Quadrics Supercomputers World Ltd., Bristol, UK. *Elan Kernel Communications Manual.*

[33] Quadrics Supercomputers World Ltd., Bristol, UK. *Elan Programming Manual.*

[34] Quadrics Supercomputers World Ltd., Bristol, UK. *Elan Reference Manual.*

[35] Quadrics Supercomputers World Ltd., Bristol, UK. *Elite Reference Manual.*

[36] A. Srivastava and A. Eustace. ATOM: A System for Building Customized Program Analysis Tools. Technical Report 94/2, WRL Research Report, March 1994.

[37] `http://www.ssfnet.org`.

[38] `http://www.acl.lanl.gov/30TeraOpRFP/SampleApps/sweep3d/sweep3d.html`.

[39] `http://www.ittc.ku.edu/utime`.

[40] Felix Wolf. *EPILOG Binary Trace-Data Format.* Forschungszentrum Julich, November 2001.